目錄

《深入解析Go》

因为自己对Go底层的东西比较感兴趣，所以抽空在写一本开源的书籍《深入解析Go》。写这本书不表示我能力很强，而是我愿意分享，和大家一起分享对Go语言的内部实现的一些研究。

我一直认为知识是用来分享的，让更多的人分享自己拥有的一切知识这个才是人生最大的快乐。

这本书目前我放在Github上，时间有限、能力有限，所以希望更多的朋友参与到这个开源项目中来。

1 如何阅读

欢迎来到Go的世界，让我们开始探索吧！

Go是一种新的语言，一种并发的、带垃圾回收的、快速编译的语言。它具有以下特点：

- 它可以在一台计算机上用几秒钟的时间编译一个大型的Go程序。
- Go为软件构造提供了一种模型，它使依赖分析更加容易，且避免了大部分C风格include文件与库的开头。
- Go是静态类型的语言，它的类型系统没有层级。因此用户不需要在定义类型之间的关系上花费时间，这样感觉起来比典型的面向对象语言更轻量级。
- Go完全是垃圾回收型的语言，并为并发执行与通信提供了基本的支持。
- 按照其设计，Go打算为多核机器上系统软件的构造提供一种方法。

Go是一种编译型语言，它结合了解释型语言的游刃有余，动态类型语言的开发效率，以及静态类型的安全性。它也打算成为现代的，支持网络与多核计算的语言。要满足这些目标，需要解决一些语言上的问题：一个富有表达能力但轻量级的类型系统，并发与垃圾回收机制，严格的依赖规范等等。这些无法通过库或工具解决好，因此Go也就应运而生了。

在本章中，我们将讲述Go的安装方法，以及如何阅读本书。

links

- 目录
- 下一节: 从源码安装Go

1.1 从源代码安装Go

本书面向的是已经对Go语言有一定的经验，希望能了解它的底层机制的用户。因此，只推荐从源代码安装Go。

Go源码安装

在Go的源代码中，有些部分是用Plan 9 C和AT&T汇编写的，因此假如你要想从源码安装，就必须安装C的编译工具。

在Mac系统中，只要你安装了Xcode，就已经包含了相应的编译工具。

在类Unix系统中，需要安装gcc等工具。例如Ubuntu系统可通过在终端中执行 `sudo apt-get install gcc libc6-dev` 来安装编译工具。

在Windows系统中，你需要安装MinGW，然后通过MinGW安装gcc，并设置相应的环境变量。

Go使用Mercurial进行版本管理，首先你必须安装了Mercurial，然后才能下载。假设你已经安装好Mercurial，执行如下代码：

假设已经位于Go的安装目录 `$GO_INSTALL_DIR` 下

```
hg clone -u release https://code.google.com/p/go
cd go/src
./all.bash
```

运行all.bash后出现"ALL TESTS PASSED"字样时才算安装成功。

上面是Unix风格的命令，Windows下的安装方式类似，只不过是运行all.bat，调用的编译器是MinGW的gcc。

然后设置几个环境变量，

```
export GOROOT=$HOME/go
export GOBIN=$GOROOT/bin
export PATH=$PATH:$GOBIN
```

看到如下图片即说明你已经安装成功



图1.1 源码安装之后执行Go命令的图

如果出现Go的Usage信息，那么说明Go已经安装成功了；如果出现该命令不存在，那么可以检查一下自己的PATH环境变中是否包含了Go的安装目录。

links

- 目录
- 上一节: 如何阅读
- 下一节: 本书的组织结构

1.2 本书的组织结构

本书结构

第二章首先会介绍一些Go的基本数据结构的实现，如slice和map。

第三章会介绍Go语言中的函数调用协议。

第四章分析runtime初始化过程。

第五章是goroutine的调度。

第六章分析Go语言中的内存管理。

第七章分析Go语言中一些高级数据结构的实现。

第八章是网络封装的实现

第九章讲cgo使用的一些技术

第十章是其它一些杂项

推荐的阅读方式

本书的写作基本上是按一个循序浅近的过程。大多数章节可以独立阅读，如内存管理，goroutine调度等。而某些知识则需要前面章节的一些基础知识，比如cgo必须了解前面函数调用协议方面的一些知识，第七章高级数据结构最好对前面内存管理和goroutine调度有一定的了解。

推荐的阅读方式还是按本文章节顺序，如果读者已经有一定基础，也可以只挑自己感兴趣的章节阅读。

如果想更深入的了解Go语言的内部实现，希望读者能拿着Go的源代码亲自分析。通过自己学习研究得到的东西才是理解最深的。

links

- 目录
- 上一节: 从源码安装Go
- 下一节: 基本技巧

1.3 基本技巧

研究Go的内部实现，这里介绍一些基本的技巧。

阅读源代码

Go语言的源代码布局是有一些规律的。假定读者在$GOROOT下:

```
- ./misc 一些工具
- ./src 源代码
- ./src/cmd 命令工具，包括6c, 6l, 6g等等。最后打包成go命令。
- ./src/pkg 各个package的源代码
- ./src/pkg/runtime Go的runtime包，本书分析的最主要的部分
-  AUTHORS — 文件，官方 Go语言作者列表
|— CONTRIBUTORS — 文件，第三方贡献者列表
|— LICENSE — 文件，Go语言发布授权协议
|— PATENTS — 文件，专利
|— README — 文件，README文件，大家懂的。提一下，经常有人说：Go官网打不开啊，怎么办？其实，在README中说到了这个。该文件还提到，如果通过二进制安装，需要设置GOROOT环境变量；如果你将Go放在了/usr/local/go中，则可以不设置该环境变量（Windows下是C:\go）。当然，建议不管什么时候都设置GOROOT。另外，确保$GOROOT/bin在PATH目录中。
|— VERSION — 文件，当前Go版本
|— api — 目录，包含所有API列表，方便IDE使用
|— doc — 目录，Go语言的各种文档，官网上有的，这里基本会有，这也就是为什么说可以本地搭建"官网"。这里面有不少其他资源，比如gopher图标之类的。
|— favicon.ico — 文件，官网logo
|— include — 目录，Go 基本工具依赖的库的头文件
|— lib — 目录，文档模板
|— misc — 目录，其他的一些工具，相当于大杂烩，大部分是各种编辑器的Go语言支持，还有cgo的例子等
|— robots.txt — 文件，搜索引擎robots文件
|— src — 目录，Go语言源码：基本工具（编译器等）、标准库
`— test — 目录，包含很多测试程序（并非_test.go方式的单元测试，而是包含main包的测试），包括一些fixbug测试。可以通过这个学到一些特性的使用。
```

学习Go语言的内部实现，主要依靠对源代码的分析，所以阅读源代码是很好的方式。linus谈到如何学习Linux内核时也说过"Read the F**ing Source code"。

使用调试器

通过gdb下断点，跟踪程序的行为。调试跟代码的方式是源代码阅读的一种辅助手段。

用户代码入口是在main.main，runtime库中的函数可以通过runtime.XXX断点捕获。比如写一个test.go：

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("hello world!")
}
```

编译，调试

```
go build test.go
gdb test
```

可以在main.main处下断点，单步执行，你会发现进入了一个runtime·convT2E的函数。这个就是由于fmt.Println接受的是一个interface，而传入的是一个string，这里会做一个转换。以这个为一个突破点去跟代码，就可以研究Go语言中具体类型如何转为interface抽象类型。

分析生成的汇编代码

有时候分析会需要研究生成的汇编代码，这里介绍生成汇编代码的方法。

```
go tool 6g -S hello.go
```

-S参数表示打印出汇编代码，更多参数可以通过-h参数查看。

```
go tool 6g -h
```

或者可以反汇编生成的可执行文件：

```
go build test.go
go tool 6l -a test | less
```

本机是amd64的机器，如果是i386的机器，则命令是8g

需要注意的是用6g的-S生成的汇编代码和6l -a生成的反汇编代码是不太一样的。前者是直接对源代码进行汇编，后者是对可执行文件进行反汇编。在6l进行链接过程中，可能会在原汇编文件基础上插入新的指令。所以6l反汇编出来的是最接近真实代码的。

不过Go的汇编语法跟常用的有点不太一致，可能读起来会不太习惯。还有另一种方式，就是在用gdb调试的过程中查看汇编。

```
gdb test
b main.main
disas
```

links

- 目录
- 上一节: 本书的组织结构
- 下一节: 基本数据结构

2 基本数据结构

这一章中我们将看一下基本的数据结构，都是Go语言内置的类型。这些知识很基础，但是理解它们非常重要。

我们将从最基本的类型开始，Go语言的基本类型部分跟C语言很类似，熟习C语言的朋友们应该不会陌生。我们也将对slice和map的实现一窥究竟。看完这章，你会知道slice不是一个指针，它在栈中是占三个机器字节的。

好吧，让我们开始吧！

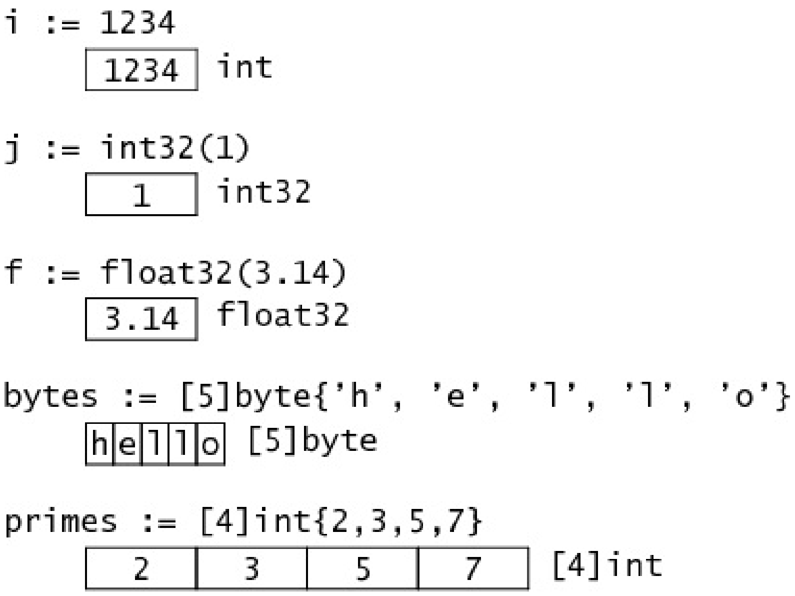
links

- 目录
- 上一节: 基本技巧
- 下一节: 基本类型

2.1 基本类型

向新手介绍Go语言时，解释一下Go中各种类型变量在内存中的布局通常有利于帮助他们加深理解。

先看一些基础的例子:



变量i属于类型int，在内存中用一个32位字长(word)表示。(32位内存布局方式）

变量j由于做了精确的转换，属于int32类型。尽管i和j有着相同的内存布局，但是它们属于不同的类型：赋值操作 `i = j` 是一种类型错误，必须写成更精确的转换方式： `i = int(j)` 。

变量f属于float类型，Go语言当前使用32位浮点型值表示(float32)。它与int32很像，但是内部实现不同。

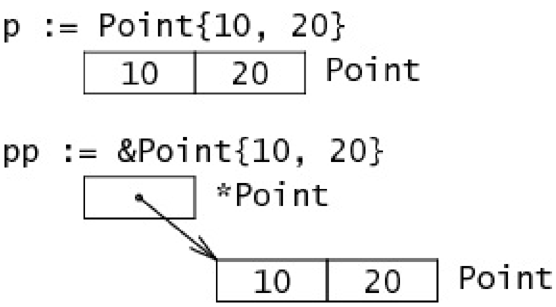
接下来，变量bytes的类型是[5]byte，一个由5个字节组成的数组。它的内存表示就是连起来的5个字节，就像C的数组。类似地，变量primes是4个int的数组。

结构体和指针

与C相同而与Java不同的是，Go语言让程序员决定何时使用指针。举例来说，这种类型定义：

```
type Point struct { X, Y int }
```

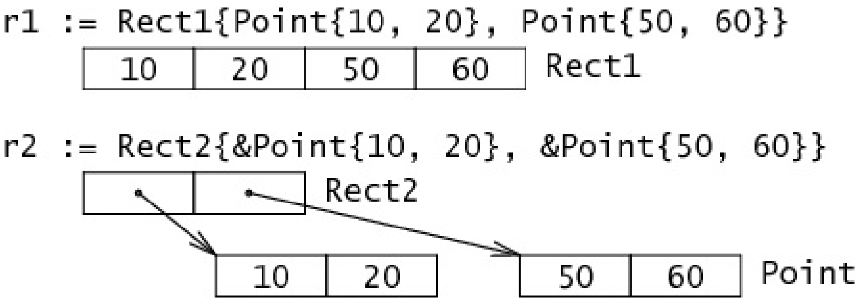
先来定义一个简单的struct类型，名为Point，表示内存中两个相邻的整数。



`Point{10,20}` 表示一个已初始化的Point类型。对它进行取地址表示一个指向刚刚分配和初始化的Point类型的指针。前者在内存中是两个词，而后者是一个指向两个词的指针。

结构体的域在内存中是紧挨着排列的。

```
type Rect1 struct { Min, Max Point }
type Rect2 struct { Min, Max *Point }
```

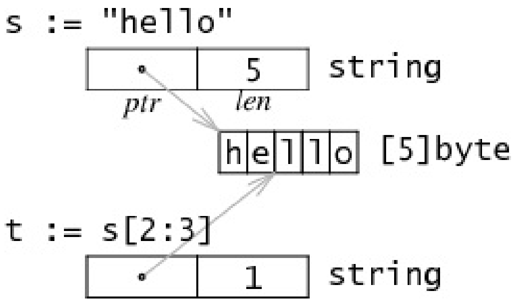


Rect1是一个具有两个Point类型属性的结构体，由在一行的两个Point--四个int代表。Rect2是一个具有两个 `*Point` 类型属性的结构体，由两个*Point表示。

使用过C的程序员可能对 `Point` 和 `*Point` 的不同毫不见怪，但用惯Java或Python的程序员们可能就不那么轻松了。Go语言给了程序员基本内存层面的控制，由此提供了诸多能力，如控制给定数据结构集合的总大小、内存分配的次数、内存访问模式以及建立优秀系统的所有要点。

字符串

有了前面的准备，我们就可以开始研究更有趣的数据类型了。



（灰色的箭头表示已经实现的但不能直接可见的指针）

字符串在Go语言内存模型中用一个2字长的数据结构表示。它包含一个指向字符串存储数据的指针和一个长度数据。因为string类型是不可变的，对于多字符串共享同一个存储数据是安全的。切分操作 `str[i:j]` 会得到一个新的2字长结构，一个可能不同的但仍指向同一个字节序列(即上文说的存储数据)的指针和长度数据。这意味着字符串切分可以在不涉及内存分配或复制操作。这使得字符串切分的效率等同于传递下标。

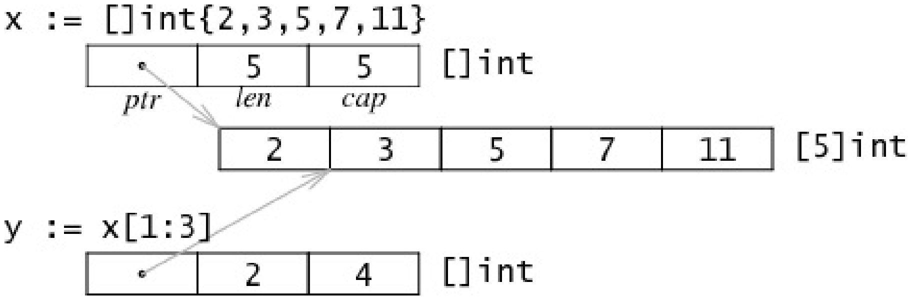
（说句题外话，在Java和其他语言里有一个有名的“疑难杂症”：在你分割字符串并保存时，对于源字符串的引用在内存中仍然保存着完整的原始字符串--即使只有一小部分仍被需要，Go也有这个“毛病”。另一方面，我们努力但又失败了的是，让字符串分割操作变得昂贵--包含一次分配和一次复制。在大多数程序中都避免了这么做。）

links

- 目录
- 上一节: 基本数据结构
- 下一节: slice

2.2 slice

一个slice是一个数组某个部分的引用。在内存中，它是一个包含3个域的结构体：指向slice中第一个元素的指针，slice的长度，以及slice的容量。长度是下标操作的上界，如x[i]中i必须小于长度。容量是分割操作的上界，如x[i:j]中j不能大于容量。



数组的slice并不会实际复制一份数据，它只是创建一个新的数据结构，包含了另外的一个指针，一个长度和一个容量数据。如同分割一个字符串，分割数组也不涉及复制操作：它只是新建了一个结构来放置一个不同的指针，长度和容量。在例子中，对 `[]int{2,3,5,7,11}` 求值操作会创建一个包含五个值的数组，并设置x的属性来描述这个数组。分割表达式 `x[1:3]` 并不分配更多的数据：它只是写了一个新的slice结构的属性来引用相同的存储数据。在例子中，长度为2--只有y[0]和y[1]是有效的索引，但是容量为4--y[0:4]是一个有效的分割表达式。

由于slice是不同于指针的多字长结构，分割操作并不需要分配内存，甚至没有通常被保存在堆中的slice头部。这种表示方法使slice操作和在C中传递指针、长度对一样廉价。Go语言最初使用一个指向以上结构的指针来表示slice，但是这样做意味着每个slice操作都会分配一块新的内存对象。即使使用了快速的分配器，还是给垃圾收集器制造了很多没有必要的工作。移除间接引用及分配操作可以让slice足够廉价，以避免传递显式索引。

slice的扩容

其实slice在Go的运行时库中就是一个C语言动态数组的实现，在$GOROOT/src/pkg/runtime/runtime.h中可以看到它的定义：

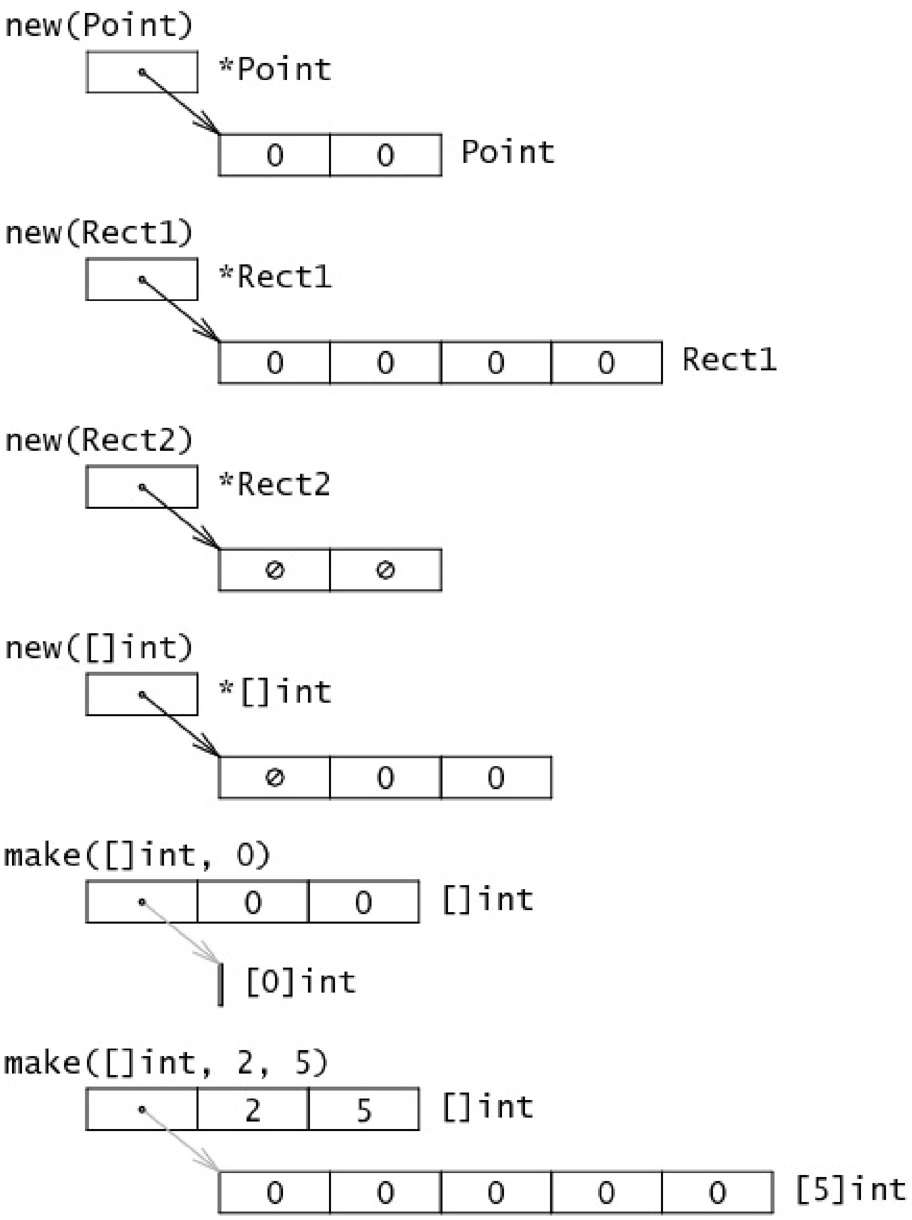
```
struct    Slice
{    // must not move anything
    byte*    array;        // actual data
    uintgo    len;        // number of elements
    uintgo    cap;        // allocated number of elements
};
```

在对slice进行append等操作时，可能会造成slice的自动扩容。其扩容时的大小增长规则是：

- 如果新的大小是当前大小2倍以上，则大小增长为新大小
- 否则循环以下操作：如果当前大小小于1024，按每次2倍增长，否则每次按当前大小1/4增长。直到增长的大小超过或等于新大小。

make和new

Go有两个数据结构创建函数：new和make。两者的区别在学习Go语言的初期是一个常见的混淆点。基本的区别是 `new(T)` 返回一个 `*T` ，返回的这个指针可以被隐式地消除引用（图中的黑色箭头）。而 `make(T, args)` 返回一个普通的T。通常情况下，T内部有一些隐式的指针（图中的灰色箭头）。一句话，new返回一个指向已清零内存的指针，而make返回一个复杂的结构。



有一种方法可以统一这两种创建方式，但是可能会与C/C++的传统有显著不同：定义`make(*T)`来返回一个指向新分配的T的指针，这样一来，new(Point)得写成make(*Point)。但这样做实在是和人们期望的分配函数太不一样了，所以Go没有采用这种设计。

slice与unsafe.Pointer相互转换

有时候可能需要使用一些比较tricky的技巧，比如利用make弄一块内存自己管理，或者用cgo之类的方式得到的内存，转换为Go类型使用。

从slice中得到一块内存地址是很容易的：

```
s := make([]byte, 200)
ptr := unsafe.Pointer(&s[0])
```

从一个内存指针构造出Go语言的slice结构相对麻烦一些，比如其中一种方式：

```
var ptr unsafe.Pointer
s := ((*[1<<10]byte)(ptr))[:200]
```

先将 `ptr` 强制类型转换为另一种指针，一个指向 `[1<<10]byte` 数组的指针，这里数组大小其实是假的。然后用slice操作取出这个数组的前200个，于是 `s` 就是一个200个元素的slice。

或者这种方式：

```
var ptr unsafe.Pointer
var s1 = struct {
    addr uintptr
    len int
    cap int
}{ptr, length, length}
s := *(*[]byte)(unsafe.Pointer(&s1))
```

把slice的底层结构写出来，将addr，len，cap等字段写进去，将这个结构体赋给s。相比上一种写法，这种更好的地方在于cap更加自然，虽然上面写法中实际上1<<10就是cap。

或者使用reflect.SliceHeader的方式来构造slice，比较推荐这种做法：

```
var o []byte
sliceHeader := (*reflect.SliceHeader)((unsafe.Pointer(&o)))
sliceHeader.Cap = length
sliceHeader.Len = length
sliceHeader.Data = uintptr(ptr)
```

links

- 目录
- 上一节: 基本类型
- 下一节: map的实现

2.3 map的实现

Go中的map在底层是用哈希表实现的，你可以在 $GOROOT/src/pkg/runtime/hashmap.goc 找到它的实现。

数据结构

哈希表的数据结构中一些关键的域如下所示：

```
struct Hmap
{
    uint8   B;    // 可以容纳2^B个项
    uint16  bucketsize;   // 每个桶的大小

    byte    *buckets;     // 2^B个Buckets的数组
    byte    *oldbuckets;  // 前一个buckets，只有当正在扩容时才不为空
};
```

上面给出的结构体只是Hmap的部分的域。需要注意到的是，这里直接使用的是Bucket的数组，而不是Bucket*指针的数组。这意味着，第一个Bucket和后面溢出链的Bucket分配有些不同。第一个Bucket是用的一段连续的内存空间，而后面溢出链的Bucket的空间是使用mallocgc分配的。

这个hash结构使用的是一个可扩展哈希的算法，由hash值mod当前hash表大小决定某一个值属于哪个桶，而hash表大小是2的指数，即上面结构体中的2^B。每次扩容，会增大到上次大小的两倍。结构体中有一个buckets和一个oldbuckets是用来实现增量扩容的。正常情况下直接使用buckets，而oldbuckets为空。如果当前哈希表正在扩容中，则oldbuckets不为空，并且buckets大小是oldbuckets大小的两倍。

具体的Bucket结构如下所示：

```
struct Bucket
{
    uint8  tophash[BUCKETSIZE]; // hash值的高8位....低位从bucket的array定位到bucket
    Bucket *overflow;           // 溢出桶链表，如果有
    byte   data[1];             // BUCKETSIZE keys followed by BUCKETSIZE values
};
```

其中BUCKETSIZE是用宏定义的8，每个bucket中存放最多8个key/value对, 如果多于8个，那么会申请一个新的bucket，并将它与之前的bucket链起来。

按key的类型采用相应的hash算法得到key的hash值。将hash值的低位当作Hmap结构体中buckets数组的index，找到key所在的bucket。将hash的高8位存储在了bucket的tophash中。注意，这里高**8**位不是用来当作**key/value**在**bucket**内部的**offset**的，而是作为一个主键，在查找时对**tophash**数组的每一项进行顺序匹配的。先比较hash值高位与bucket的tophash[i]是否相等，如果相等则再比较bucket的第i个的key与所给的key是否相等。如果相等，则返回其对应的value，反之，在overflow buckets中按照上述方法继续寻找。

整个hash的存储如下图所示(临时先采用了XX同学画的图，这个图有点问题)：

图2.2 HMap的存储结构

注意一个细节是Bucket中key/value的放置顺序，是将keys放在一起，values放在一起，为什么不将key和对应的value放在一起呢？如果那么做，存储结构将变成key1/value1/key2/value2… 设想如果是这样的一个map[int64]int8，考虑到字节对齐，会浪费很多存储空间。不得不说通过上述的一个小细节，可以看出Go在设计上的深思熟虑。

增量扩容

大家都知道哈希表表就是以空间换时间，访问速度是直接跟填充因子相关的，所以当哈希表太满之后就需要进行扩容。

如果扩容前的哈希表大小为2^B，扩容之后的大小为2^(B+1)，每次扩容都变为原来大小的两倍，哈希表大小始终为2的指数倍，则有(hash mod 2^B)等价于(hash & (2^B-1))。这样可以简化运算，避免了取余操作。

假设扩容之前容量为X，扩容之后容量为Y，对于某个哈希值hash，一般情况下(hash mod X)不等于(hash mod Y)，所以扩容之后要重新计算每一项在哈希表中的新位置。当hash表扩容之后，需要将那些旧的pair重新哈希到新的table上(源代码中称之为evacuate)， 这个工作并没有在扩容之后一次性完成，而是逐步的完成（在insert和remove时每次搬移1-2个pair），Go语言使用的是增量扩容。

为什么会增量扩容呢？主要是缩短map容器的响应时间。假如我们直接将map用作某个响应实时性要求非常高的web应用存储，如果不采用增量扩容，当map里面存储的元素很多之后，扩容时系统就会卡往，导致较长一段时间内无法响应请求。不过增量扩容本质上还是将总的扩容时间分摊到了每一次哈希操作上面。

扩容会建立一个大小是原来2倍的新的表，将旧的bucket搬到新的表中之后，并不会将旧的bucket从oldbucket中删除，而是加上一个已删除的标记。

正是由于这个工作是逐渐完成的，这样就会导致一部分数据在old table中，一部分在new table中， 所以对于hash table的insert, remove, lookup操作的处理逻辑产生影响。只有当所有的bucket都从旧表移到新表之后，才会将oldbucket释放掉。

扩容的填充因子是多少呢？如果grow的太频繁，会造成空间的利用率很低， 如果很久才grow，会形成很多的overflow buckets，查找的效率也会下降。 这个平衡点如何选取呢(在go中，这个平衡点是有一个宏控制的(#define LOAD 6.5), 它的意思是这样的，如果table中元素的个数大于table中能容纳的元素的个数， 那么就触发一次grow动作。那么这个6.5是怎么得到的呢？原来这个值来源于作者的一个测试程序，遗憾的是没能找到相关的源码，不过作者给出了测试的结果：

```
        LOAD    %overflow  bytes/entry     hitprobe    missprobe
        4.00         2.13        20.77         3.00         4.00
        4.50         4.05        17.30         3.25         4.50
        5.00         6.85        14.77         3.50         5.00
        5.50        10.55        12.94         3.75         5.50
        6.00        15.27        11.67         4.00         6.00
        6.50        20.90        10.79         4.25         6.50
        7.00        27.14        10.15         4.50         7.00
        7.50        34.03         9.73         4.75         7.50
        8.00        41.10         9.40         5.00         8.00

 %overflow   = percentage of buckets which have an overflow bucket
 bytes/entry = overhead bytes used per key/value pair
 hitprobe    = # of entries to check when looking up a present key
 missprobe   = # of entries to check when looking up an absent key
```

可以看出作者取了一个相对适中的值。

查找过程

1. 根据key计算出hash值。
2. 如果存在old table, 首先在old table中查找，如果找到的bucket已经evacuated，转到步骤3。 反之，返回其对应的value。
3. 在new table中查找对应的value。

这里一个细节需要注意一下。不认真看可能会以为低位用于定位bucket在数组的index，那么高位就是用于key/valule在bucket内部的offset。事实上高8位不是用作offset的，而是用于加快key的比较的。

```
do { //对每个桶b
    //依次比较桶内的每一项存放的tophash与所求的hash值高位是否相等
    for(i = 0, k = b->data, v = k + h->keysize * BUCKETSIZE; i < BUCKETSIZE; i++, k += h->keysize, v += h->valuesize)
 {
        if(b->tophash[i] == top) {
            k2 = IK(h, k);
            t->key->alg->equal(&eq, t->key->size, key, k2);
            if(eq) { //相等的情况下再去做key比较...
                *keyp = k2;
                return IV(h, v);
            }
        }
    }
    b = b->overflow; //b设置为它的下一下溢出链
} while(b != nil);
```

插入过程分析

1. 根据key算出hash值，进而得出对应的bucket。
2. 如果bucket在old table中，将其重新散列到new table中。
3. 在bucket中，查找空闲的位置，如果已经存在需要插入的key，更新其对应的value。
4. 根据table中元素的个数，判断是否grow table。
5. 如果对应的bucket已经full，重新申请新的bucket作为overbucket。
6. 将key/value pair插入到bucket中。

这里也有几个细节需要注意一下。

在扩容过程中，oldbucket是被冻结的，查找时会在oldbucket中查找，但不会在oldbucket中插入数据。如果在oldbucket是找到了相应的key，做法是将它迁移到新bucket后加入evalucated标记。并且还会额外的迁移另一个pair。

然后就是只要在某个bucket中找到第一个空位，就会将key/value插入到这个位置。也就是位置位于bucket前面的会覆盖后面的(类似于存储系统设计中做删除时的常用的技巧之一，直接用新数据追加方式写，新版本数据覆盖老版本数据)。找到了相同的key或者找到第一个空位就可以结束遍历了。不过这也意味着做删除时必须完全的遍历bucket所有溢出链，将所有的相同key数据都删除。所以目前map的设计是为插入而优化的，删除效率会比插入低一些。

map设计中的性能优化

读完map源代码发现作者还是做了很多设计上的选择的。本人水平有限，谈不上优劣的点评，这里只是拿出来与读者分享。

HMap中是Bucket的数组，而不是Bucket指针的数组。好的方面是可以一次分配较大内存，减少了分配次数，避免多次调用mallocgc。但相应的缺点，其一是可扩展哈希的算法并没有发生作用，扩容时会造成对整个数组的值拷贝(如果实现上用Bucket指针的数组就是指针拷贝了，代价小很多)。其二是首个bucket与后面产生了不一致性。这个会使删除逻辑变得复杂一点。比如删除后面的溢出链可以直接删除，而对于首个bucket，要等到evalucated完毕后，整个oldbucket删除时进行。

没有重用设freelist重用删除的结点。作者把这个加了一个TODO的注释，不过想了一下觉得这个做的意义不大。因为一方面，bucket大小并不一致，重用比较麻烦。另一方面，下层存储已经做过内存池的实现了，所以这里不做重用也会在内存分配那一层被重用的，

bucket直接key/value和间接key/value优化。这个优化做得蛮好的。注意看代码会发现，如果key或value小于128字节，则它们的值是直接使用的bucket作为存储的。否则bucket中存储的是指向实际key/value数据的指针，

bucket存8个key/value对。查找时进行顺序比较。第一次发现高位居然不是用作offset，而是用于加快比较的。定位到bucket之后，居然是一个顺序比较的查找过程。后面仔细想了想，觉得还行。由于bucket只有8个，顺序比较下来也不算过分。仍然是O(1)只不过前面系数大一点点罢了。相当于hash到一个小范围之后，在这个小范围内顺序查找。

插入删除的优化。前面已经提过了，插入只要找到相同的key或者第一个空位，bucket中如果存在一个以上的相同key，前面覆盖后面的(只是如果，实际上不会发生)。而删除就需要遍历完所有bucket溢出链了。这样map的设计就是为插入优化的。考虑到一般的应用场景，这个应该算是很合理的。

作者还列了另个2个TODO：将多个几乎要empty的bucket合并；如果table中元素很少，考虑shrink table。(毕竟现在的实现只是单纯的grow)。

links

- 目录
- 上一节: slice
- 下一节: nil的语义

2.4 nil的语义

什么？nil是一种数据结构么？为什么会讲到它，没搞错吧？没搞错。不仅仅是Go语言中，每门语言中nil都是非常重要的，它代表的是空值的语义。

在不同语言中，表示空这个概念都有细微不同。比如在scheme语言(一种lisp方言)中，nil是true的！而在ruby语言中，一切都是对象，连nil也是一个对象！在C中NULL跟0是等价的。

按照Go语言规范，任何类型在未初始化时都对应一个零值：布尔类型是false，整型是0，字符串是""，而指针，函数，interface，slice，channel和map的零值都是nil。

interface

一个interface在没有进行初始化时，对应的值是nil。也就是说 `var v interface{}` ，

此时v就是一个nil。在底层存储上，它是一个空指针。与之不同的情况是，interface值为空。比如：

```
var v *T
var i interface{}
i = v
```

此时i是一个interface，它的值是nil，但它自身不为nil。

Go中的error其实就是一个实现了Error方法的接口：

```
type error interface {
    Error() string
}
```

因此，我们可以自定义一个error：

```
type Error struct {
    errCode uint8
}
func (e *Error) Error() string {
        switch e.errCode {
        case 1:
                return "file not found"
        case 2:
                return "time out"
        case 3:
                return "permission denied"
        default:
                return "unknown error"
         }
}
```

如果我们这样使用它：

```
func checkError(err error) {
    if err != nil {
        panic(err)
    }
}
var e *Error
checkError(e)
```

e是nil的，但是当我们checkError时就会panic。请读者思考一下为什么？

总之，interface跟C语言的指针一样非常灵活，关于空的语义，也跟空指针一样容易困扰新手的，需要注意。

string和slice

string的空值是""，它是不能跟nil比较的。即使是空的string，它的大小也是两个机器字长的。slice也类似，它的空值并不是一个空指针，而是结构体中的指针域为空，空的slice的大小也是三个机器字长的。

channel和map

channel跟string或slice有些不同，它在栈上只是一个指针，实际的数据都是由指针所指向的堆上面。

跟channel相关的操作有：初始化/读/写/关闭。channel未初始化值就是nil，未初始化的channel是不能使用的。下面是一些操作规则：

- 读或者写一个nil的channel的操作会永远阻塞。
- 读一个关闭的channel会立刻返回一个channel元素类型的零值。
- 写一个关闭的channel会导致panic。

map也是指针，实际数据在堆中，未初始化的值是nil。

links

- 目录
- 上一节: map的实现
- 下一节: 函数调用协议

3 函数调用协议

理解Go的函数调用协议对于研究其内部实现非常重要。这里将会介绍Go进行函数调用时的内存布局，参数传递和返回值的约定。正如C和汇编都是同一套约定所以能相互调用一样，Go和C以及汇编也是要满足某些约定才能够相互调用。

本章先从Go调用C和汇编的例子开始(非cgo方式)，通过分析其实现学习Go的函数调用协议。然后将会研究go和defer关键字等神奇的魔法。接着会研究连续栈的实现，最后看一下闭包。

这一章的内容将是后面研究cgo，goroutine实现的基础。连续栈技术是Go能够开千千万万条“线程”而不耗尽内存的基本保证，也为cgo带来了很大的限制，这些将会在后面章节中再讨论。

好，让我们进入正题吧！

links

- 目录
- 上一节: nil的语义
- 下一节: Go调用汇编和C

3.1 Go调用汇编和C

只要不使用C的标准库函数，Go中是可以直接调用C和汇编语言的。其实道理很简单，Go的运行时库就是用C和汇编实现的，Go必须是能够调用到它们的。当然，会有一些额外的约束，这就是函数调用协议。

Go中调用汇编

假设我们做一个汇编版本的加法函数。首先GOPATH的src下新建一个add目录，然后在该目录加入add.go的文件，内容如下：

```
package add

func Add(a, b uint64) uint64 {
    return a+b
}
```

这个函数将两个uint64的数字相加，并返回结果。我们写一个简单的函数调用它，内容如下：

```
package main

import (
  "fmt"
  "add"
)

func main() {
     fmt.Println(add.Add(2, 15))
}
```

可以看到输出了结果为17。好的，接下来让我们删除Add函数的实现，只留下定义部分：

```
package add

func Add(a, b uint64) uint64
```

然后在add.go同一目录中建立一个add_amd64.s的文件(假设你使用的是64位系统)，内容如下：

```
TEXT    ·Add+0(SB),$0-24
MOVQ    a+0(FP),BX
MOVQ    b+8(FP),BP
ADDQ    BP,BX
MOVQ    BX,res+16(FP)
RET     ,
```

虽然汇编是相当难理解的，但我相信读懂上面这段不会有困难。前两条MOVQ指令分别将第一个参数放到寄存器BX，第二个参数放到寄存器BP，然后ADDQ指令将两者相加后，最后的MOVQ和RET指令返回结果。

现在，再次运行前面的main函数，它将使用自定义的汇编版本函数，可以看到成功的输出了结果17。从这个例子中可以看出Go是可以直接调用汇编实现的函数的。大多时候不必要你去写汇编，即使是研究Go的内部实现，能读懂汇编已经很足够了。

也许你真的觉得在Go中写汇编很酷，但是不要忽视了这些忠告：

- 汇编很难编写，特别是很难写好。通常编译器会比你写出更快的代码。
- 汇编仅能运行在一个平台上。在这个例子中，代码仅能运行在 amd64 上。这个问题有一个解决方案是给 Go 对于 x86 和 不同版本的代码分别写一套代码，文件名相应的以_386.s和_arm.s结尾。
- 汇编让你和底层绑定在一起，而标准的 Go 不会。例如，slice 的长度当前是 32 位整数。但是也不是不可能为长整型。

> 当发生这些变化时，这些代码就被破坏了。

当前Go编译器不能将汇编编译为函数的内联，但是对于小的Go函数是可以的。因此使用汇编可能意味着让你的程序更慢。

有时需要汇编给你带来一些力量（不论是性能方面的原因，还是一些相当特殊的关于CPU的操作）。对于什么时候应该使用它，Go源码包括了若干相当好的例子（可以看看 crypto 和 math）。由于它非常容易实践，所以这绝对是个学习汇编的好途径。

Go中调用C

接下来，我们继续尝试在Go中调用C，跟调用汇编的过程很类似。首先删掉前面的add_amd64.s文件，并确保add.go文件中只是给出了Add函数的声明部分：

```
package add

func Add(a, b uint64) uint64
```

然后在add.go同目录中，新建一个add.c文件，内容如下：

```
#include "runtime.h"

void ·Add(uint64 a, uint64 b, uint64 ret) {
    ret = a + b;
    FLUSH(&ret);
}
```

编译该包，运行前面的测试函数：

```
go install add
```

会发现输出结果为17，说明Go中成功地调用到了C写的函数。

要注意的是不管是C或是汇编实现的函数，其函数名都是以·开头的。还有，C文件中需要包含runtime.h头文件。这个原因在该文件中有说明： Go用了特殊寄存器来存放像全局的struct G和struct M。包含这个头文件可以让所有链接到Go的C文件都知道这一点，这样编译器可以避免使用这些特定的寄存器作其它用途。

让我们仔细看一下这个C实现的函数。可以看到函数的返回值为空，而参数多了一个，第三个参数实际上被作为了返回值使用。其中FLUSH是在pkg/runtime/runtime.h中定义为USED(x)，这个定义是Go的C编译器自带的primitive，作用是抑制编译器优化掉对*x的赋值的。如果你很好奇USED是怎样定义的，可以去$GOROOT/include/libc.h文件里去找找。

被调函数中对参数ret的修改居然返回到了调用函数，这个看起来似乎不可理解，不过早期的C编译器确实是可以这么做的。

函数调用时的内存布局

Go中使用的C编译器其实是plan9的C编译器，和我们平时理解的gcc等会有一些区别。我们将上面的add.c汇编一下：

```
go tool 6c -I $GOROOT/src/pkg/runtime -S add.c
```

生成的汇编代码大概是这个样子的：

```
"".Add t=1 size=16 value=0 args=0x18 locals=0
000000 00000 (add.c:3)    TEXT    "".Add+0(SB),4,$0-24
000000 00000 (add.c:3)    NOP    ,
000000 00000 (add.c:3)    NOP    ,
000000 00000 (add.c:3)    FUNCDATA    $2,gcargs.0<>+0(SB)
000000 00000 (add.c:3)    FUNCDATA    $3,gclocals.1<>+0(SB)
000000 00000 (add.c:4)    MOVQ    a+8(FP),AX
0x0005 00005 (add.c:4)    ADDQ    b+16(FP),AX
0x000a 00010 (add.c:4)    MOVQ    AX,c+24(FP)
0x000f 00015 (add.c:5)    RET    ,
000000 48 8b 44 24 08 48 03 44 24 10 48 89 44 24 18 c3  H.D$.H.D$.H.D$..
```

这是Go使用的汇编代码，是一种类似plan9的汇编代码。类似a+8(FP)这种表示的含义是“变量名+偏移(寄存器)”。其中FP是帧寄存器，它是一个伪寄存器，实际上是内存位置的一个引用，其实就是BP(栈基址寄存器)上移一个机器字长位置的内存地址。

函数调用之前，a+8(FP),b+16(FP)分别表示参数a和b，而参数3的位置被空着，在被调函数中，这个位置将用于存放返回值。此时的其内存布局如下所示：

```
参数3
参数2
参数1  <-SP
```

进入被调函数之后，内存布局如下所示：

```
参数3
参数2
参数1  <-FP
保存PC <-SP
...
...
```

CALL指令会使得SP下移，SP位置的内存用于保存返回地址。帧寄存器FP此时位置在SP上面。在plan9汇编中，进入函数之后的前几条指令并没有出现 `push ebp; mov esp ebp` 这种模式。plan9函数调用协议中采用的是caller-save的模式，也就是由调用者负责保存寄存器。注意这和传统的C是不同的。传统C中是callee-save的模式，被调函数要负责保存它想使用的寄存器，在函数退出时恢复这些寄存器。

需要注意的是参数和返回值都是有对齐的。这里是按Structrnd对齐的，Structrnd在源代码中义为sizeof(uintptr)。

links

- 目录
- 上一节: 函数调用协议
- 下一节: 多值返回

3.2 多值返回

Go语言是支持多值返回的。怎么实现的呢？让我们先看一看C语言是如果返回多个值的。在C中如果想返回多个值，通常会在调用函数中分配返回值的空间，并将返回值的指针传给被调函数。

```
int ret1, ret2;
f(a, b, &ret1, &ret2)
```

被调函数被定义为下面形式，在函数中会修改ret1和ret2。对指针参数所指向的内容的修改会被返回到调用函数，用这种方式实现多值返回。

```
void f(int arg1, int arg2, int *ret1, int *ret2);
```

所以，从表面上看Go的多值返回只不过像是这种实现方式的一个语法糖衣。其实简单的这么理解也没什么影响，但实际上Go不是这么干的，Go和我们常用的C编译器的函数调用协议是不同的。

假设我们定义一个Go函数如下：

```
func f(arg1, arg2 int) (ret1, ret2 int)
```

Go的做法是在传入的参数之上留了两个空位，被调者直接将返回值放在这两空位，函数f调用前其内存布局是这样的：

```
为ret2保留空位
为ret1保留空位
参数3
参数2
参数1  <-SP
```

调用之后变为：

```
为ret2保留空位
为ret1保留空位
参数2
参数1  <-FP
保存PC <-SP
f的栈
...
```

Go的C编译器按是plan9的C编译器实现的，在被调函数中对参数值的修改是会返回到调用函数中的。在函数体中设置ret1和ret2的值，实际上会被编译成这样：

```
MOVQ    BX,ret1+16(FP)
...
MOVQ    BX,ret2+24(FP)
```

对ret1+16(FP)的赋值其实是修改的调用函数的栈中的内容，这样就会将结果返回给调用函数了。这就是Go和C函数调用协议中很重要的一个区别：为了实现多值返回，Go是使用栈空间来返回值的。而常见的C语言是通过寄存器来返回值的。

links

- 目录
- 上一节: Go调用汇编和C

- 下一节: go关键字

3.2 go关键字

在Go语言中，表达式go f(x, y, z)会启动一个新的goroutine运行函数f(x, y, z)。函数f，变量x、y、z的值是在原goroutine计算的，只有函数f的执行是在新的goroutine中的。显然，新的goroutine不能和当前go线程用同一个栈，否则会相互覆盖。所以对go关键字的调用协议与普通函数调用是不同的。

首先，让我们看一下如果是C代码新建一条线程的实现会是什么样子的。大概会先建一个结构体，结构体里存f、x、y和z的值。然后写一个help函数，将这个结构体指针作为输入，函数体内调用f(x, y, z)。接下来，先填充结构体，然后调用newThread(help, structptr)。其中help是刚刚那个函数，它会调用f(x, y, z)。help函数将作为所有新建线程的入口函数。

这样做有什么问题么？没什么问题...只是这样实现代价有点高，每次调用都会花上不少的指令。其实Go语言中对go关键字的实现会更加hack一些，避免了这么做。

先看看正常的函数调用，下面是调用f(1, 2, 3)时的汇编代码：

```
MOVL    $1, 0(SP)
MOVL    $2, 4(SP)
MOVL    $3, 8(SP)
CALL    f(SB)
```

首先将参数1、2、3进栈，然后调用函数f。

下面是go f(1, 2, 3)生成的代码：

```
MOVL    $1, 0(SP)
MOVL    $2, 4(SP)
MOVL    $3, 8(SP)
PUSHQ   $f(SB)
PUSHQ   $12
CALL    runtime.newproc(SB)
POPQ    AX
POPQ    AX
```

对比一个会发现，前面部分跟普通函数调用是一样的，将参数存储在正常的位置，并没有新建一个辅助的结构体。接下来的两条指令有些不同，将f和12作为参数进栈而不直接调用f，然后调用函数 `runtime.newproc` 。

12是参数占用的大小。 `runtime.newproc` 函数接受的参数分别是：参数大小，新的goroutine是要运行的函数，函数的n个参数。

在 `runtime.newproc` 中，会新建一个栈空间，将栈参数的12个字节拷贝到新栈空间中并让栈指针指向参数。这时的线程状态有点像当被调度器剥夺CPU后一样，寄存器PC、SP会被保存到类似于进程控制块的一个结构体struct G内。f被存放在了struct G的entry域，后面进行调度器恢复goroutine的运行，新线程将从f开始执行。

和前面说的如果用C实现的差别就在于，没有使用辅助的结构体，而 `runtime.newproc` 实际上就是help函数。在函数协议上，go表达式调用就比普通的函数调用多四条指令而已，并且在实际上并没有为go关键字设计一套特殊的东西。不得不说这个做法真的非常精妙！

总结一个，go关键字的实现仅仅是一个语法糖衣而已，也就是：

```
go f(args)
```

可以看作

```
runtime.newproc(size, f, args)
```

links

- 目录
- 上一节: 多值返回
- 下一节: defer关键字

3.4 defer关键字

defer和go一样都是Go语言提供的关键字。defer用于资源的释放，会在函数返回之前进行调用。一般采用如下模式：

```
f,err := os.Open(filename)
if err != nil {
    panic(err)
}
defer f.Close()
```

如果有多个defer表达式，调用顺序类似于栈，越后面的defer表达式越先被调用。

不过如果对defer的了解不够深入，使用起来可能会踩到一些坑，尤其是跟带命名的返回参数一起使用时。在讲解defer的实现之前先看一看使用defer容易遇到的问题。

defer使用时的坑

先来看看几个例子。例1：

```
func f() (result int) {
    defer func() {
        result++
    }()
    return 0
}
```

例2：

```
func f() (r int) {
     t := 5
     defer func() {
       t = t + 5
     }()
     return t
}
```

例3：

```
func f() (r int) {
    defer func(r int) {
          r = r + 5
    }(r)
    return 1
}
```

请读者先不要运行代码，在心里跑一遍结果，然后去验证。

例1的正确答案不是0，例2的正确答案不是10，如果例3的正确答案不是6......

defer是在return之前执行的。这个在 官方文档 中是明确说明了的。要使用defer时不踩坑，最重要的一点就是要明白，**return xxx**这一条语句并不是一条原子指令!

函数返回的过程是这样的：先给返回值赋值，然后调用defer表达式，最后才是返回到调用函数中。

defer表达式可能会在设置函数返回值之后，在返回到调用函数之前，修改返回值，使最终的函数返回值与你想象的不一致。

其实使用defer时，用一个简单的转换规则改写一下，就不会迷糊了。改写规则是将return语句拆成两句写，return xxx会被改写成:

```
返回值 = xxx
调用defer函数
空的return
```

先看例1，它可以改写成这样：

```
func f() (result int) {
     result = 0  //return语句不是一条原子调用，return xxx其实是赋值＋ret指令
     func() { //defer被插入到return之前执行，也就是赋返回值和ret指令之间
         result++
     }()
     return
}
```

所以这个返回值是1。

再看例2，它可以改写成这样：

```
func f() (r int) {
     t := 5
     r = t //赋值指令
     func() {        //defer被插入到赋值与返回之间执行，这个例子中返回值r没被修改过
         t = t + 5
     }
     return        //空的return指令
}
```

所以这个的结果是5。

最后看例3，它改写后变成：

```
func f() (r int) {
     r = 1  //给返回值赋值
     func(r int) {        //这里改的r是传值传进去的r，不会改变要返回的那个r值
          r = r + 5
     }(r)
     return        //空的return
}
```

所以这个例子的结果是1。

defer确实是在return之前调用的。但表现形式上却可能不像。本质原因是return xxx语句并不是一条原子指令，defer被插入到了赋值 与 ret之间，因此可能有机会改变最终的返回值。

defer的实现

defer关键字的实现跟go关键字很类似，不同的是它调用的是runtime.deferproc而不是runtime.newproc。

在defer出现的地方，插入了指令call runtime.deferproc，然后在函数返回之前的地方，插入指令call runtime.deferreturn。

普通的函数返回时，汇编代码类似：

```
add xx SP
return
```

如果其中包含了defer语句，则汇编代码是：

```
call runtime.deferreturn，
add xx SP
return
```

goroutine的控制结构中，有一张表记录defer，调用runtime.deferproc时会将需要defer的表达式记录在表中，而在调用runtime.deferreturn的时候，则会依次从defer表中出栈并执行。

links

- 目录
- 上一节: go关键字
- 下一节: 连续栈

3.5 连续栈

Go语言支持goroutine，每个goroutine需要能够运行，所以它们都有自己的栈。假如每个goroutine分配固定栈大小并且不能增长，太小则会导致溢出，太大又会浪费空间，无法存在许多的goroutine。

为了解决这个问题，goroutine可以初始时只给栈分配很小的空间，然后随着使用过程中的需要自动地增长。这就是为什么Go可以开千千万万个goroutine而不会耗尽内存。

Go1.3版本之后则使用的是continuous stack，下面将具体分析一下这种技术。

基本原理

每次执行函数调用时Go的runtime都会进行检测，若当前栈的大小不够用，则会触发"中断"，从当前函数进入到Go的运行时库，Go的运行时库会保存此时的函数上下文环境，然后分配一个新的足够大的栈空间，将旧栈的内容拷贝到新栈中，并做一些设置，使得当函数恢复运行时，函数会在新分配的栈中继续执行，仿佛整个过程都没发生过一样，这个函数会觉得自己使用的是一块大小"无限"的栈空间。

实现过程

在研究Go的实现细节之前让我们先自己思考一下应该如何实现。第一步肯定要有某种机制检测到当前栈大小不够用了，这个应该是把当前的栈寄存器SP跟栈的可用栈空间的边界进行比较。能够检测到栈大小不够用，就相当于捕捉到了"中断"。

捕获完"中断"，第二步要做的，就应该是进入运行时，保存当前goroutine的上下文。别陷入如何保存上下文的细节，先假如我们把函数栈增长时的上下文保存好了，那下一步就是分配新的栈空间了，我们可以将分配空间想象成就是调用一下malloc而已。

接下来怎么办呢？我们要将旧栈中的内容拷贝到新栈中，然后让函数继续在新栈中运行。这里先暂时忽略旧栈内容拷贝到新栈中的一些技术难点，假设在新栈空间中恢复了"中断"时的上下文，从运行时返回到函数。

函数在新的栈中继续运行了，但是还有个问题：函数如何返回。因为函数返回后栈是要缩小的，否则就会内存浪费空间了，所以还需要在函数返回时处理栈缩小的问题。

具体细节

如何捕获到函数的栈空间不足

Go语言和C不同，不是使用栈指针寄存器和栈基址寄存器确定函数的栈的。在Go的运行时库中，每个goroutine对应一个结构体G，大致相当于进程控制块的概念。这个结构体中存了stackbase和stackguard，用于确定这个goroutine使用的栈空间信息。每个Go函数调用的前几条指令，先比较栈指针寄存器跟g->stackguard，检测是否发生栈溢出。如果栈指针寄存器值超越了stackguard就需要扩展栈空间。

为了加深理解，下面让我们跟踪一下代码，并看看实际生成的汇编吧。首先写一个test.go文件，内容如下：

```
package main
func main() {
    main()
}
```

然后生成汇编文件：

```
go tool 6g -S test.go | head -8
```

可以看以输出是：

```
000000 00000 (test.go:3)    TEXT    "".main+0(SB),$0-0
000000 00000 (test.go:3)    MOVQ    (TLS),CX
0x0009 00009 (test.go:3)    CMPQ    SP,(CX)
0x000c 00012 (test.go:3)    JHI    ,21
0x000e 00014 (test.go:3)    CALL    ,runtime.morestack00_noctxt(SB)
0x0013 00019 (test.go:3)    JMP    ,0
0x0015 00021 (test.go:3)    NOP    ,
```

让我们好好看一下这些指令。(TLS)取到的是结构体G的第一个域，也就是g->stackguard地址，将它赋值给CX。然后CX地址的值与SP进行比较，如果SP大于g->stackguard了，则会调用runtime.morestack函数。这几条指令的作用就是检测栈是否溢出。

不过并不是所有函数在链接时都会插入这种指令。如果你读源代码，可能会发现 `#pragma textflag 7` ，或者在汇编函数中看到 `TEXT reuntime.exit(SB),7,$0` ，这种函数就是不会检测栈溢出的。这个是编译标记，控制是否生成栈溢出检测指令。

runtime.morestack是用汇编实现的，做的事情大致是将一些信息存在M结构体中，这些信息包括当前栈桢，参数，当前函数调用，函数返回地址（两个返回地址，一个是runtime.morestack的函数地址，一个是f的返回地址）。通过这些信息可以把新栈和旧栈链起来。

```
void runtime.morestack() {
    if(g == g0) {
        panic();
    } else {
        m->morebuf.gobuf_pc = getCallerCallerPC();
        void *SP = getCallerSP();
        m->morebuf.gobuf_sp = SP;
        m->moreargp = SP;
        m->morebuf.gobuf_g = g;
        m->morepc = getCallerPC();

        void *g0 = m->g0;
        g = g0;
        setSP(g0->g_sched.gobuf_sp);
        runtime.newstack();
    }
}
```

需要注意的就是newstack是切换到m->g0的栈中去调用的。m->g0是调度器栈，go的运行时库的调度器使用的都是m->g0。

旧栈数据复制到新栈

runtime.morestack会调用于runtime.newstack，newstack做的事情很好理解：分配一个足够大的新的空间，将旧的栈中的数据复制到新的栈中，进行适当的修饰，伪装成调用过runtime.lessstack的样子（这样当函数返回时就会调用runtime.lessstack再次进入runtime中做一些栈收缩的处理）。

这里有一个技术难点：旧栈数据复制到新栈的过程，要考虑指针失效问题。

比如有某个指针，引用了旧栈中的地址，如果仅仅是将旧栈内容搬到新栈中，那么该指针就失效了，因为旧栈已被释放，应该修改这个指针让它指向新栈的对应地址。考虑如下代码：

```
func f1() {
    var a A
    f(&a)
}
func f2(a *A) {
    // modify a
}
```

如果在f2中发生了栈增长，此时分配更大的空间作为新栈，并将旧栈内容拷贝到新栈中，仅仅这样是不够的，因为f2中的a还是指向旧栈中的f1的，所以必须调整。

Go实现了精确的垃圾回收，运行时知道每一块内存对应的对象的类型信息。在复制之后，会进行指针的调整。具体做法是，对当前栈帧之前的每一个栈帧，对其中的每一个指针，检测指针指向的地址，如果指向地址是落在旧栈范围内的，则将它加上一个偏移使它指向新栈的相应地址。这个偏移值等于新栈基地址减旧栈基地址。

runtime.lessstack比较简单，它其实就是切换到m->g0栈之后调用runtime.oldstack函数。这时之前保存的那个Stktop结构体是时候发挥作用了，从上面可以找到旧栈空间的SP和PC等信息，通过runtime.gogo跳转过去，整个过程就完成了。

```
gp = m->curg; //当前g
top = (Stktop*)gp->stackbase; //取得Stktop结构体
label = top->gobuf; //从结构体中取出Gobuf
runtime·gogo(&label, cret); //通过Gobuf恢复上下文
```

小结

1. 使用分段栈的函数头几个指令检测SP和stackguard，调用runtime.morestack
2. runtime.morestack函数的主要功能是保存当前的栈的一些信息，然后转换成调度器的栈调用runtime.newstack
3. runtime.newstack函数的主要功能是分配空间，装饰此空间，将旧的frame和arg弄到新空间
4. 使用gogocall的方式切换到新分配的栈，gogocall使用的JMP返回到被中断的函数
5. 继续执行遇到RET指令时会返回到runtime.lessstack，lessstack做的事情跟morestack相反，它要准备好从new stack到old stack

整个过程有点像一次中断，中断处理时保存当时的现场，弄个新的栈，中断恢复时恢复到新栈中运行。栈的收缩是垃圾回收的过程中实现的．当检测到栈只使用了不到1/4时，栈缩小为原来的1/2.

links

- 目录
- 上一节: defer关键字
- 下一节: 闭包的实现

3.6 闭包的实现

闭包是由函数及其相关引用环境组合而成的实体(即：闭包=函数+引用环境)。

Go中的闭包

闭包是函数式语言中的概念，没有研究过函数式语言的用户可能很难理解闭包的强大，相关的概念超出了本书的范围。Go语言是支持闭包的，这里只是简单地讲一下在Go语言中闭包是如何实现的。

```
func f(i int) func() int {
    return func() int {
        i++
        return i
    }
}
```

函数f返回了一个函数，返回的这个函数，返回的这个函数就是一个闭包。这个函数中本身是没有定义变量i的，而是引用了它所在的环境（函数f）中的变量i。

```
c1 := f(0)
c2 := f(0)
c1()    // reference to i, i = 0, return 1
c2()    // reference to another i, i = 0, return 1
```

c1跟c2引用的是不同的环境，在调用i++时修改的不是同一个i，因此两次的输出都是1。函数f每进入一次，就形成了一个新的环境，对应的闭包中，函数都是同一个函数，环境却是引用不同的环境。

变量i是函数f中的局部变量，假设这个变量是在函数f的栈中分配的，是不可以的。因为函数f返回以后，对应的栈就失效了，f返回的那个函数中变量i就引用一个失效的位置了。所以闭包的环境中引用的变量不能够在栈上分配。

escape analyze

在继续研究闭包的实现之前，先看一看Go的一个语言特性：

```
func f() *Cursor {
    var c Cursor
    c.X = 500
    noinline()
    return &c
}
```

Cursor是一个结构体，这种写法在C语言中是不允许的，因为变量c是在栈上分配的，当函数f返回后c的空间就失效了。但是，在Go语言规范中有说明，这种写法在Go语言中合法的。语言会自动地识别出这种情况并在堆上分配c的内存，而不是函数f的栈上。

为了验证这一点，可以观察函数f生成的汇编代码：

```
MOVQ    $type."".Cursor+0(SB),(SP)    // 取变量c的类型，也就是Cursor
PCDATA    $0,$16
PCDATA    $1,$0
CALL    ,runtime.new(SB)    // 调用new函数，相当于new(Cursor)
PCDATA    $0,$-1
MOVQ    8(SP),AX    // 取c.X的地址放到AX寄存器
MOVQ    $500,(AX)    // 将AX存放的内存地址的值赋为500
MOVQ    AX,"".~r0+24(FP)
ADDQ    $16,SP
```

识别出变量需要在堆上分配，是由编译器的一种叫escape analyze的技术实现的。如果输入命令：

```
go build --gcflags=-m main.go
```

可以看到输出：

```
./main.go:20: moved to heap: c
./main.go:23: &c escapes to heap
```

表示c逃逸了，被移到堆中。escape analyze可以分析出变量的作用范围，这是对垃圾回收很重要的一项技术。

闭包结构体

回到闭包的实现来，前面说过，闭包是函数和它所引用的环境。那么是不是可以表示为一个结构体呢：

```
type Closure struct {
    F func()()
    i *int
}
```

事实上，Go在底层确实就是这样表示一个闭包的。让我们看一下汇编代码：

```
func f(i int) func() int {
    return func() int {
        i++
        return i
    }
}


MOVQ    $type.int+0(SB),(SP)
PCDATA    $0,$16
PCDATA    $1,$0
CALL    ,runtime.new(SB)    // 是不是很熟悉，这一段就是i = new(int)
...
MOVQ    $type.struct { F uintptr; A0 *int }+0(SB),(SP)    // 这个结构体就是闭包的类型
...
CALL    ,runtime.new(SB)    // 接下来相当于 new(Closure)
PCDATA    $0,$-1
MOVQ    8(SP),AX
NOP    ,
MOVQ    $"".func·001+0(SB),BP
MOVQ    BP,(AX)                // 函数地址赋值给Closure的F部分
NOP    ,
MOVQ    "".&i+16(SP),BP        // 将堆中new的变量i的地址赋值给Closure的值部分
MOVQ    BP,8(AX)
MOVQ    AX,"".~r1+40(FP)
ADDQ    $24,SP
RET    ,
```

其中func·001是另一个函数的函数地址，也就是f返回的那个函数。

小结

1. Go语言支持闭包
2. Go语言能通过escape analyze识别出变量的作用域，自动将变量在堆上分配。将闭包环境变量在堆上分配是Go实现闭包的基础。
3. 返回闭包时并不是单纯返回一个函数，而是返回了一个结构体，记录下函数返回地址和引用的环境中的变量地址。

links

- 目录
- 上一节: 连续栈
- 下一节: Go语言程序初始化过程

4 Go语言程序初始化过程

作为下一章goroutine调度的一个前序，本章先讲一些基础内容，看一看Go语言编写的程序的初始化过程。其实初始化过程中会做很多很多的事情，这里忽略大部分细节，只看一下脉络。从程序入口开始分析也是学习源代码的一个好方式。

首先，写一个hello world文件，内容如下：

```
package main
import "fmt"
func main() {
	fmt.Println("hello world!")
}
```

编译，使用gdb调试。给下列函数下断点：

```
_rt0_amd64_darwin
main
_rt0_amd64
runtime.check
runtime.args
runtime.osinit
runtime.hashinit
runtime.schedinit
runtime.newproc
runtime.mstart
main.main
runtime.exit
```

你可能需要根据自己的系统将_rt0_amd64_darwin改成_rt0_amd64_linux或者别的。在gdb中先点r，回车，然后点c，回车，接着一路回车。

别着急，只是让你有一个直观的感受一下Go程序从系统初始化直到退出必经的流程。下面让我们正式开始吧！

links

- 目录
- 上一节: 闭包的实现
- 下一节: 系统初始化

4.1 系统初始化

整个程序启动是从_rt0_amd64_darwin开始的，然后JMP到main，接着到_rt0_amd64。前面只有一点点汇编代码，做的事情就是通过参数argc和argv等，确定栈的位置，得到寄存器。下面将从_rt0_amd64开始分析。

这里首先会设置好m->g0的栈，将当前的SP设置为stackbase，将SP往下大约64K的地方设置为stackguard。然后会获取处理器信息，放在全局变量runtime·cpuid_ecx和runtime·cpuid_edx中。接着，设置本地线程存储。本地线程存储是依赖于平台实现的，比如说这台机器上是调用操作系统函数thread_fast_set_cthread_self。设置本地线程存储之后还会立即测试一下，写入一个值再读出来看是否正常。

本地线程存储

这里解释一下本地线程存储。比如说每个goroutine都有自己的控制信息，这些信息是存放在一个结构体G中。假设我们有一个全局变量g是结构体G的指针，我们希望只有唯一的全局变量g，而不是g0，g1，g2...但是我们又希望不同goroutine去访问这个全局变量g得到的并不是同一个东西，它们得到的是相对自己线程的结构体G，这种情况下就需要本地线程存储。g确实是一个全局变量，却在不同线程有多份不同的副本。每个goroutine去访问g时，都是对应到自己线程的这一份副本。

设置好本地线程存储之后，就可以为每个goroutine和machine设置寄存器了。这样设置好了之后，每次调用get_tls(r)，就会将当前的goroutine的g的地址放到寄存器r中。你可以在源代码中看到一些类似这样的汇编：

```
get_tls(CX)
MOVQ    g(CX), AX //get_tls(CX)之后，g(CX)得到的就是当前的goroutine的g
```

不同的goroutine调用 `get_tls` ，得到的g是本地的结构体G的，结构体中记录goroutine的相关信息。

初始化顺序

接下来的事情就非常直白，可以直接上代码：

```
CLD                // convention is D is always left cleared
CALL    runtime·check(SB) //检测像int8,int16,float等是否是预期的大小，检测cas操作是否正常
MOVL    16(SP), AX        // copy argc
MOVL    AX, 0(SP)
MOVQ    24(SP), AX        // copy argv
MOVQ    AX, 8(SP)
CALL    runtime·args(SB)    //将argc,argv设置到static全局变量中了
CALL    runtime·osinit(SB)    //osinit做的事情就是设置runtime.ncpu，不同平台实现方式不一样
CALL    runtime·hashinit(SB)    //使用读/dev/urandom的方式从内核获得随机数种子
CALL    runtime·schedinit(SB)    //内存管理初始化，根据GOMAXPROCS设置使用的procs等等
```

proc.c中有一段注释，也说明了bootstrap的顺序：

```
// The bootstrap sequence is:
//
// call osinit
// call schedinit
// make & queue new G
// call runtime·mstart
//
// The new G calls runtime·main.
```

先调用osinit，再调用schedinit，创建就绪队列并新建一个G，接着就是mstart。这几个函数都不太复杂。

调度器初始化

让我们看一下runtime.schedinit函数。该函数其实是包装了一下其它模块的初始化函数。有调用mallocinit，mcommoninit分别对内存管理模块初始化，对当前的结构体M初始化。

接着调用runtime.goargs和runtime.goenvs，将程序的main函数参数argc和argv等复制到了os.Args中。

也是在这个函数中，根据环境变量GOMAXPROCS决定可用物理线程数目的：

```
procs = 1;
p = runtime·getenv("GOMAXPROCS");
if(p != nil && (n = runtime·atoi(p)) > 0) {
    if(n > MaxGomaxprocs)
        n = MaxGomaxprocs;
    procs = n;
}
```

回到前面的汇编代码继续看：

```
// 新建一个G，当它运行时会调用main.main
PUSHQ    $runtime·main·f(SB)        // entry
PUSHQ    $0             // arg size
CALL    runtime·newproc(SB)
POPQ    AX
POPQ    AX

// start this M
CALL    runtime·mstart(SB)
```

还记得前面章节讲的go关键字的调用协议么？先将参数进栈，再被调函数指针和参数字节数进栈，接着调用runtime.newproc函数。所以这里其实就是新开个goroutine执行runtime.main。

runtime.newproc会把runtime.main放到就绪线程队列里面。本线程继续执行runtime.mstart，m意思是machine。runtime.mstart会调用到调度函数schedule

schedule函数绝不返回，它会根据当前线程队列中线程状态挑选一个来运行。由于当前只有这一个goroutine，它会被调度，然后就到了runtime.main函数中来，runtime.main会调用用户的main函数，即main.main从此进入用户代码。前面已经写过helloworld了，用gdb调试，一步一步的跟踪观察这个过程。

links

- 目录
- 上一节: Go语言程序初始化过程
- 下一节: main.main之前的准备

4.2 main.main之前的准备

main.main就是用户的main函数。这里是指Go的runtime在进入用户main函数之前做的一些事情。

前面已经介绍了从Go程序执行后的第一条指令，到启动runtime.main的主要流程，比如其中要设置好本地线程存储，设置好main函数参数，根据环境变量GOMAXPROCS设置好使用的procs，初始化调度器和内存管理等等。

接下来将是从runtime.main到main.main之间的一些过程。注意，main.main是在runtime.main函数里面调用的。不过在调用main.main之前，还有一些工作要做。

sysmon

在main.main执行之前，Go语言的runtime库会初始化一些后台任务，其中一个任务就是sysmon。

```
newm(sysmon, nil);
```

newm新建一个结构体M，第一个参数是这个结构体M的入口函数，也就说会在一个新的物理线程中运行sysmon函数。由此可见sysmon是一个地位非常高的后台任务，整个函数体一个死循环的形式，目前主要处理两个事件：对于网络的epoll以及抢占式调度的检测。大致过程如下：

```
for(;;) {
    runtime.usleep(delay);
    if(lastpoll != 0 && lastpoll + 10*1000*1000 > now) {
        runtime.netpoll();
    }
    retake(now);    // 根据每个P的状态和运行时间决定是否要进行抢占
}
```

sysmon会根据系统当前的繁忙程度睡一小段时间，然后每隔10ms至少进行一次epoll并唤醒相应的goroutine。同时，它还会检测是否有P长时间处于Psyscall状态或Prunning状态，并进行抢占式调度。

scavenger

scavenger是另一个后台任务，但是它的创建跟sysmon有点区别：

```
runtime·newproc(&scavenger, nil, 0, 0, runtime·main);
```

newproc创建一个goroutine，第一个参数是goroutine运行的函数。scavenger的地位是没有sysmon那么高的——sysmon是由物理线程运行的，而scavenger只是由goroutine运行的。接下来的章节会说明goroutine与物理线程的区别。

那么，scavenger执行什么工作？它又为什么不像sysmon那样呢？其实scavenger执行的是runtime·MHeap_Scavenger函数。它将一些不再使用的内存归还给操作系统。Go是一门`垃圾回收`的语言，垃圾回收会在系统运行过程中被触发，内存会被归还到Go的内存管理系统中，Go的内存管理是基于内存池进行重用的，而这个函数会真正地将内存归还给操作系统。

scavenger显然没有sysmon要求那么高，所以它仅仅是一个普通的goroutine而不是一个线程。

main.main在这些后台任务运行起来之后执行，不过在它执行之前，还有最后一个：main.init，每个包的init函数会在包使用之前先执行。

links

- 目录
- 上一节: 系统初始化

* 下一节: goroutine调度

5. goroutine调度

links

- 目录
- 上一节: main.main之前的准备
- 下一节: 调度器相关数据结构

5.1 调度器相关数据结构

Go的调度的实现，涉及到几个重要的数据结构。运行时库用这几个数据结构来实现goroutine的调度，管理goroutine和物理线程的运行。这些数据结构分别是结构体G，结构体M，结构体P，以及Sched结构体。前三个的定义在文件runtime/runtime.h中，而Sched的定义在runtime/proc.c中。Go语言的调度相关实现也是在文件proc.c中。

结构体G

G是goroutine的缩写，相当于操作系统中的进程控制块，在这里就是goroutine的控制结构，是对goroutine的抽象。其中包括goid是这个goroutine的ID，status是这个goroutine的状态，如Gidle,Grunnable,Grunning,Gsyscall,Gwaiting,Gdead等。

```
struct G
{
    uintptr    stackguard;    // 分段栈的可用空间下界
    uintptr    stackbase;    // 分段栈的栈基址
    Gobuf    sched;        //进程切换时，利用sched域来保存上下文
    uintptr    stack0;
    FuncVal*    fnstart;        // goroutine运行的函数
    void*    param;        // 用于传递参数，睡眠时其它goroutine设置param，唤醒时此goroutine可以获取
    int16    status;        // 状态Gidle,Grunnable,Grunning,Gsyscall,Gwaiting,Gdead
    int64    goid;        // goroutine的id号
    G*    schedlink;
    M*    m;        // for debuggers, but offset not hard-coded
    M*    lockedm;    // G被锁定只能在这个m上运行
    uintptr    gopc;    // 创建这个goroutine的go表达式的pc
    ...
};
```

结构体G中的部分域如上所示。可以看到，其中包含了栈信息stackbase和stackguard，有运行的函数信息fnstart。这些就足够成为一个可执行的单元了，只要得到CPU就可以运行。

goroutine切换时，上下文信息保存在结构体的sched域中。goroutine是轻量级的`线程`或者称为`协程`，切换时并不必陷入到操作系统内核中，所以保存过程很轻量。看一下结构体G中的Gobuf，其实只保存了当前栈指针，程序计数器，以及goroutine自身。

```
struct Gobuf
{
    // The offsets of these fields are known to (hard-coded in) libmach.
    uintptr    sp;
    byte*    pc;
    G*    g;
    ...
};
```

记录g是为了恢复当前goroutine的结构体G指针，运行时库中使用了一个常驻的寄存器`extern register G* g`，这个是当前goroutine的结构体G的指针。这样做是为了快速地访问goroutine中的信息，比如，Go的栈的实现并没有使用%ebp寄存器，不过这可以通过g->stackbase快速得到。"extern register"是由6c，8c等实现的一个特殊的存储。在ARM上它是实际的寄存器；其它平台是由段寄存器进行索引的线程本地存储的一个槽位。在linux系统中，对g和m使用的分别是0(GS)和4(GS)。需要注意的是，链接器还会根据特定操作系统改变编译器的输出，例如，6l/linux下会将0(GS)重写为-16(FS)。每个链接到Go程序的C文件都必须包含runtime.h头文件，这样C编译器知道避免使用专用的寄存器。

结构体M

M是machine的缩写，是对机器的抽象，每个m都是对应到一条操作系统的物理线程。M必须关联了P才可以执行Go代码，但是当它处理阻塞或者系统调用中时，可以不需要关联P。

```
struct M
{
    G*    g0;        // 带有调度栈的goroutine
    G*    gsignal;    // signal-handling G 处理信号的goroutine
    void    (*mstartfn)(void);
    G*    curg;        // M中当前运行的goroutine
    P*    p;        // 关联P以执行Go代码 (如果没有执行Go代码则P为nil)
    P*    nextp;
    int32    id;
    int32    mallocing; //状态
    int32    throwing;
    int32    gcing;
    int32    locks;
    int32    helpgc;        //不为0表示此m在做帮忙gc。helpgc等于n只是一个编号
    bool    blockingsyscall;
    bool    spinning;
    Note    park;
    M*    alllink;    // 这个域用于链接allm
    M*    schedlink;
    MCache    *mcache;
    G*    lockedg;
    M*    nextwaitm;    // next M waiting for lock
    GCStats    gcstats;
    ...
};
```

这里也是截取结构体M中的部分域。和G类似，M中也有alllink域将所有的M放在allm链表中。lockedg是某些情况下，G锁定在这个M中运行而不会切换到其它M中去。M中还有一个MCache，是当前M的内存的缓存。M也和G一样有一个常驻寄存器变量，代表当前的M。同时存在多个M，表示同时存在多个物理线程。

结构体M中有两个G是需要关注一下的，一个是curg，代表结构体M当前绑定的结构体G。另一个是g0，是带有调度栈的goroutine，这是一个比较特殊的goroutine。普通的goroutine的栈是在堆上分配的可增长的栈，而g0的栈是M对应的线程的栈。所有调度相关的代码，会先切换到该goroutine的栈中再执行。

结构体P

Go1.1中新加入的一个数据结构，它是Processor的缩写。结构体P的加入是为了提高Go程序的并发度，实现更好的调度。M代表OS线程。P代表Go代码执行时需要的资源。当M执行Go代码时，它需要关联一个P，当M为idle或者在系统调用中时，它也需要P。有刚好GOMAXPROCS个P。所有的P被组织为一个数组，在P上实现了工作流窃取的调度器。

```
struct P
{
    Lock;
    uint32    status;  // Pidle或Prunning等
    P*    link;
    uint32    schedtick;   // 每次调度时将它加一
    M*    m;    // 链接到它关联的M (nil if idle)
    MCache*    mcache;

    G*    runq[256];
    int32    runqhead;
    int32    runqtail;

    // Available G's (status == Gdead)
    G*    gfree;
    int32    gfreecnt;
    byte    pad[64];
};
```

结构体P中也有相应的状态：

```
Pidle,
Prunning,
Psyscall,
Pgcstop,
Pdead,
```

注意，跟G不同的是，P不存在 `waiting` 状态。MCache被移到了P中，但是在结构体M中也还保留着。在P中有一个Grunnable的goroutine队列，这是一个P的局部队列。当P执行Go代码时，它会优先从自己的这个局部队列中取，这时可以不用加锁，提高了并发度。如果发现这个队列空了，则去其它P的队列中拿一半过来，这样实现工作流窃取的调度。这种情况下是需要给调用器加锁的。

Sched

Sched是调度实现中使用的数据结构，该结构体的定义在文件proc.c中。

```
struct Sched {
    Lock;

    uint64    goidgen;

    M*    midle;     // idle m's waiting for work
    int32    nmidle;     // number of idle m's waiting for work
    int32    nmidlelocked; // number of locked m's waiting for work
    int3    mcount;     // number of m's that have been created
    int32    maxmcount;    // maximum number of m's allowed (or die)

    P*    pidle;  // idle P's
    uint32    npidle;  //idle P的数量
    uint32    nmspinning;

    // Global runnable queue.
    G*    runqhead;
    G*    runqtail;
    int32    runqsize;

    // Global cache of dead G's.
    Lock    gflock;
    G*    gfree;

    int32    stopwait;
    Note    stopnote;
    uint32    sysmonwait;
    Note    sysmonnote;
    uint64    lastpoll;

    int32    profilehz;    // cpu profiling rate
}
```

大多数需要的信息都已放在了结构体M、G和P中，Sched结构体只是一个壳。可以看到，其中有M的idle队列，P的idle队列，以及一个全局的就绪的G队列。Sched结构体中的Lock是非常必须的，如果M或P等做一些非局部的操作，它们一般需要先锁住调度器。

links

- 目录
- 上一节: goroutine调度
- 下一节: goroutine的生老病死

5.2 goroutine的生老病死

本小节将通过goroutine的创建，消亡，阻塞和恢复等过程，来观察Go语言的调度策略，这里就称之为生老病死吧。整个Go语言的调度系统是比较复杂的，为了避免结构体M和结构体P引入的其它干扰，这里主要将注意力集中到结构体G中，以goroutine为主线。

goroutine的创建

前面讲函数调用协议时说过go关键字最终被弄成了runtime.newproc。这就是一个goroutine的出生，所有新的goroutine都是通过这个函数创建的。

runtime.newproc(size, f, args)功能就是创建一个新的g，这个函数不能用分段栈，因为它假设参数的放置顺序是紧接着函数f的（见前面函数调用协议一章，有关go关键字调用时的内存布局）。分段栈会破坏这个布局，所以在代码中加入了标记#pragma textflag 7表示不使用分段栈。它会调用函数newproc1，在newproc1中可以使用分段栈。真正的工作是调用newproc1完成的。newproc1进行下面这些动作。

首先，它会检查当前结构体M中的P中，是否有可用的结构体G。如果有，则直接从中取一个，否则，需要分配一个新的结构体G。如果分配了新的G，需要将它挂到runtime的相关队列中。

获取了结构体G之后，将调用参数保存到g的栈，将sp，pc等上下文环境保存在g的sched域，这样整个goroutine就准备好了，整个状态和一个运行中的goroutine被中断时一样，只要等分配到CPU，它就可以继续运行。

```
newg->sched.sp = (uintptr)sp;
newg->sched.pc = (byte*)runtime·goexit;
newg->sched.g = newg;
runtime·gostartcallfn(&newg->sched, fn);
newg->gopc = (uintptr)callerpc;
newg->status = Grunnable;
newg->goid = runtime·xadd64(&runtime·sched.goidgen, 1);
```

然后将这个“准备好”的结构体G挂到当前M的P的队列中。这里会给予新的goroutine一次运行的机会，即：如果当前的P的数目没有到上限，也没有正在自旋抢CPU的M，则调用wakep将P立即投入运行。

wakep函数唤醒P时，调度器会试着寻找一个可用的M来绑定P，必要的时候会新建M。让我们看看新建M的函数newm：

```
// 新建一个m，它将以调用fn开始，或者是从调度器开始
static void
newm(void(*fn)(void), P *p)
{
    M *mp;
    mp = runtime·allocm(p);
    mp->nextp = p;
    mp->mstartfn = fn;
    runtime·newosproc(mp, (byte*)mp->g0->stackbase);
}
```

runtime.newm功能跟newproc相似,前者分配一个goroutine,而后者分配一个M。其实一个M就是一个操作系统线程的抽象，可以看到它会调用runtime.newosproc。

总算看到了从Go的运行时库到操作系统的接口，runtime.newosproc(平台相关的)会调用系统的runtime.clone(平台相关的)来新建一个线程，新的线程将以runtime.mstart为入口函数。runtime.newosproc是个很有意思的函数，还有一些信号处理方面的细节，但是对鉴于我们是专注于调度方面，就不对它进行更细致的分析了，感兴趣的读者可以自行去runtime/os_linux.c看看源代码。runtime.clone是用汇编实现的,代码在sys_linux_amd64.s。

既然线程是以runtime.mstart为入口的，那么接下来看mstart函数。

mstart是runtime.newosproc新建的系统线程的入口地址，新线程执行时会从这里开始运行。新线程的执行和goroutine的执行是两个概念，由于有m这一层对机器的抽象，是m在执行g而不是线程在执行g。所以线程的入口是mstart，g的执行要到schedule才算入口。函数mstart最后调用了schedule。

终于到了schedule了！

如果是从mstart进入到schedule的，那么schedule中逻辑非常简单，大概就这几步：

```
找到一个等待运行的g
如果g是锁定到某个M的，则让那个M运行
否则，调用execute函数让g在当前的M中运行
```

execute会恢复newproc1中设置的上下文，这样就跳转到新的goroutine去执行了。从newproc出生一直到运行的过程分析，到此结束!

虽然按这样a调用b，b调用c，c调用d，d调用e的方式去分析源代码谁看都会晕掉，但还是要重复一遍这里的读代码过程，希望感兴趣的读者可以拿着注释过的源码按顺序走一遍：

newproc -> newproc1 -> (如果P数目没到上限)wakep -> startm -> (可能引发)newm -> newosproc -> (线程入口)mstart -> schedule -> execute -> goroutine运行

进出系统调用

假设goroutine"生病"了，它要进入系统调用了，暂时无法继续执行。进入系统调用时，如果系统调用是阻塞的，goroutine会被剥夺CPU，将状态设置成Gsyscall后放到就绪队列。Go的syscall库中提供了对系统调用的封装，它会在真正执行系统调用之前先调用函数.entersyscall，并在系统调用函数返回后调用.exitsyscall函数。这两个函数就是通知Go的运行时库这个goroutine进入了系统调用或者完成了系统调用，调度器会做相应的调度。

比如syscall包中的Open函数，它会调用Syscall(SYS_OPEN, uintptr(unsafe.Pointer(_p0)), uintptr(mode), uintptr(perm))实现。这个函数是用汇编写的，在syscall/asm_linux_amd64.s中可以看到它的定义：

```
TEXT    ·Syscall(SB),7,$0
    CALL    runtime·entersyscall(SB)
    MOVQ    16(SP), DI
    MOVQ    24(SP), SI
    MOVQ    32(SP), DX
    MOVQ    $0, R10
    MOVQ    $0, R8
    MOVQ    $0, R9
    MOVQ    8(SP), AX    // syscall entry
    SYSCALL
    CMPQ    AX, $0xfffffffffffff001
    JLS    ok
    MOVQ    $-1, 40(SP)    // r1
    MOVQ    $0, 48(SP)    // r2
    NEGQ    AX
    MOVQ    AX, 56(SP)  // errno
    CALL    runtime·exitsyscall(SB)
    RET
ok:
    MOVQ    AX, 40(SP)    // r1
    MOVQ    DX, 48(SP)    // r2
    MOVQ    $0, 56(SP)    // errno
    CALL    runtime·exitsyscall(SB)
    RET
```

可以看到它进系统调用和出系统调用时分别调用了runtime.entersyscall和runtime.exitsyscall函数。那么，这两个函数做什么特殊的处理呢？

首先，将函数的调用者的SP,PC等保存到结构体G的sched域中。同时，也保存到g->gcsp和g->gcpc等，这个是跟垃圾回收相关的。

然后检查结构体Sched中的sysmonwait域，如果不为0，则将它置为0，并调用runtime·notewakeup(&runtime·sched.sysmonnote)。做这这一步的原因是，目前这个goroutine要进入Gsyscall状态了，它将要让出CPU。如果有人在等待CPU的话，会通知并唤醒等待者，马上就有CPU可用了。

接下来，将m的MCache置为空，并将m->p->m置为空，表示进入系统调用后结构体M是不需要MCache的，并且P也被剥离了，将P的状态设置为PSyscall。

有一个与entersyscall函数稍微不同的函数叫entersyscallblock，它会告诉提示这个系统调用是会阻塞的，因此会有一点点区别。它调用的releasep和handoffp。

releasep将P和M完全分离，使p->m为空，m->p也为空，剥离m->mcache，并将P的状态设置为Pidle。注意这里的区别，在非阻塞的系统调用entersyscall中只是设置成Psyscall，并且也没有将m->p置为空。

handoffp切换P。将P从处于syscall或者locked的M中，切换出来交给其它M。每个P中是挂了一个可执行的G的队列的，如果这个队列不为空，即如果P中还有G需要执行，则调用startm让P与某个M绑定后立刻去执行，否则将P挂到idlep队列中。

出系统调用时会调用到runtime·exitsyscall，这个函数跟进系统调用做相反的操作。它会先检查当前m的P和它状态，如果P不空且状态为Psyscall，则说明是从一个非阻塞的系统调用中返回的，这时是仍然有CPU可用的。因此将p->m设置为当前m，将p的mcache放回到m，恢复g的状态为Grunning。否则，它是从一个阻塞的系统调用中返回的，因此之前m的P已经完全被剥离了。这时会查看调用中是否还有idle的P，如果有，则将它与当前的M绑定。

如果从一个阻塞的系统调用中出来，并且出来的这一时刻又没有idle的P了，要怎么办呢？这种情况代码当前的goroutine无法继续运行了，调度器会将它的状态设置为Grunnable，将它挂到全局的就绪G队列中，然后停止当前m并调用schedule函数。

goroutine的消亡以及状态变化

goroutine的消亡比较简单，注意在函数newproc1，设置了fnstart为goroutine执行的函数，而将新建的goroutine的sched域的pc设置为了函数runtime.exit。当fnstart函数执行完返回时，它会返回到runtime.exit中。这时Go就知道这个goroutine要结束了，runtime.exit中会做一些回收工作，会将g的状态设置为Gdead等，并将g挂到P的free队列中。

从以上的分析中，其实已经基本上经历了goroutine的各种状态变化。在newproc1中新建的goroutine被设置为Grunnable状态，投入运行时设置成Grunning。在entersyscall的时候goroutine的状态被设置为Gsyscall，到出系统调用时根据它是从阻塞系统调用中出来还是非阻塞系统调用中出来，又会被设置成Grunning或者Grunnable的状态。在goroutine最终退出的runtime.exit函数中，goroutine被设置为Gdead状态。

等等，好像缺了什么？是的，Gidle始终没有出现过。这个状态好像实际上没有被用到。只有一个runtime.park函数会使goroutine进入到Gwaiting状态，但是park这个有什么作用我暂时还没看懂...

goroutine的状态变迁图：

links

- 目录
- 上一节: 调度器相关数据结构
- 下一节: 设计与演化

5.3 设计与演化

其实讲一个东西，讲它是什么样是不足够的。如果能讲清楚它为什么会是这样子，则会举一反三。为了理解goroutine的本质，这里将从最基本的线程池讲起，谈谈Go调度设计背后的故事，讲清楚它为什么是这样子。

线程池

先看一些简单点的吧。一个常规的 线程池+任务队列 的模型如图所示：

把每个工作线程叫worker的话，每条线程运行一个worker，每个worker做的事情就是不停地从队列中取出任务并执行：

```
while(!empty(queue)) {
    q = get(queue); //从任务队列中取一个(涉及加锁等)
    q->callback(); //执行该任务
}
```

当然，这是最简单的情形，但是一个很明显的问题就是一个进入callback之后，就失去了控制权。因为没有一个调度器层的东西，一个任务可以执行很长很长时间一直占用的worker线程，或者阻塞于io之类的。

也许用Go语言表述会更地道一些。好吧，那么让我们用Go语言来描述。假设我们有一些“任务”，任务是一个可运行的东西，也就是只要满足Run函数，它就是一个任务。所以我们就把这个任务叫作接口G吧。

```
type G interface {
    Run()
}
```

我们有一个全局的任务队列，里面包含很多可运行的任务。线程池的各个线程从全局的任务队列中取任务时，显然是需要并发保护的，所以有下面这个结构体：

```
type Sched struct {
    allg  []G
    lock    *sync.Mutex
}
```

以及它的变量

```
var sched Sched
```

每条线程是一个worker，这里我们给worker换个名字，就把它叫M吧。前面已经说过了，worker做的事情就是不停的去任务队列中取一个任务出来执行。于是用Go语言大概可以写成这样子：

```
func M() {
    for {
        sched.lock.Lock()    //互斥地从就绪G队列中取一个g出来运行
        if sched.allg > 0 {
            g := sched.allg[0]
            sched.allg = sched.allg[1:]
            sched.lock.Unlock()
            g.Run()        //运行它
        } else {
            sched.lock.Unlock()
        }
    }
}
```

接下来，将整个系统启动：

```
for i:=0; i<GOMAXPROCS; i++ {
    go M()
}
```

假定我们有一个满足G接口的main，然后它在自己的Run中不断地将新的任务挂到sched.allg中，这个线程池+任务队列的系统模型就会一直运行下去。

可以看到，这里在代码取中故意地用Go语言中的G，M，甚至包括GOMAXPROCS等取名字。其实本质上，Go语言的调度层无非就是这样一个工作模式的：几条物理线程，不停地取goroutine运行。

系统调用

上面的情形太简单了，就是工作线程不停地取goroutine运行，这个还不能称之为调度。调度之所以为调度，是因为有一些复杂的控制机制，比如哪个goroutine应该被运行，它应该运行多久，什么时候将它换出来。用前面的代码来说明Go的调度会有一些小问题。Run函数会一直执行，在它结束之前不会返回到调用器层面。那么假设上面的任务中Run进入到一个阻塞的系统调用了，那么M也就跟着一起阻塞了，实际工作的线程就少了一个，无法充分利用CPU。

一个简单的解决办法是在进入系统调用之前再制造一个M出来干活，这样就填补了这个进入系统调用的M的空缺，始终保证有GOMAXPROCS个工作线程在干活了。

```
func entersyscall() {
    go M()
}
```

那么出系统调用时怎么办呢？如果让M接着干活，岂不超过了GOMAXPROCS个线程了？所以这个M不能再干活了，要限制干活的M个数为GOMAXPROCS个，多了则让它们闲置(物理线程比CPU多很多就没意义了，让它们相互抢CPU反而会降低利用率)。

```
func exitsyscall() {
    if len(allm) >= GOMAXPROCS {
        sched.lock.Lock()
        sched.allg = append(sched.allg, g)    //把g放回到队列中
        sched.lock.Unlock()
        time.Sleep()    //这个M不再干活
    }
}
```

于是就变成了这样子:

其实这个也很好理解，就像线程池做负载调节一样，当任务队列很长后，忙不过来了，则再开几条线程出来。而如果任务队列为空了，则可以释放一些线程。

协程与保存上下文

大家都知道阻塞于系统调用，会白白浪费CPU。而使用异步事件或回调的思维方式又十分反人类。上面的模型既然这么简单明了，为什么不这么用呢？其实上面的东西看上去简单，但实现起来确不那么容易。

将一个正在执行的任务yield出去，再在某个时刻再弄回来继续运行，这就涉及到一个麻烦的问题，即保存和恢复运行时的上下文环境。

在此先引入协程的概念。协程是轻量级的线程，它相对线程的优势就在于协程非常轻量级，进行切换以及保存上下文环境代价非常的小。协程的具体的实现方式有多种，上面就是其中一种基于线程池的实现方式。每个协程是一个任务，可以保存和恢复任务运行时的上下文环境。

协程一类的东西一般会提供类似yield的函数。协程运行到一定时候就主动调用yield放弃自己的执行，把自己再次放回到任务队列中等待下一次调用时机等等。

其实Go语言中的goroutine就是协程。每个结构体G中有一个sched域就是用于保存自己上下文的。这样，这种goroutine就可以被换出去，再换进来。这种上下文保存在用户态完成，不必陷入到内核，非常的轻量，速度很快。保存的信息很少，只有当前的PC,SP等少量信息。只是由于要优化，所以代码看上去更复杂一些，比如要重用内存空间所以会有gfree和mhead之类的东西。

Go1.0

在前面的代码中，线程与M是直接对应的关系，这个解耦还是不够。Go1.0中将M抽出来成为了一个结构体，startm函数是线程的入口地址，而goroutine的入口地址是go表达式中的那个函数。总体上跟上面的结构差不多，进出系统调用的时候goroutine会跟M一起进入到系统调用中，schedule中会匹配g和m，让空闲的m来运行g。如果检测到干活的数量少于GOMAXPROCS并且没有空闲着的m，则会创建新的m来运行g。出系统调用的时候，如果已经有GOMAXPROCS个m在干活了，则这个出系统调用的m会被挂起，它的g也会被挂到待运行的goroutine队列中。

在Go语言中m是machine的缩写，也就是机器的抽象。它被设计成了可以运行所有的G。比如说一个g开始在某个m上运行，经过几次进出系统调用之后，可能运行它的m挂起了，其它的m会将它从队列中取出并继续运行。

每次调度都会涉及对g和m等队列的操作，这些全局的数据在多线程情况下使用就会涉及到大量的锁操作。在频繁的系统调用中这将是一个很大的开销。为了减少系统调用开销，Go1.0在这里做了一些优化的。1.0版中，在它的Sched结构体中有一个atomic字段，类型是一个volatile的无符32位整型。

```
// sched中的原子字段是一个原子的uint32，存放下列域
// 15位 mcpu  --正在占用cpu运行的m数量 (进入syscall的m是不占用cpu的)
// 15位 mcpumax  --最大允许这么多个m同时使用cpu
// 1位  waitstop  --有g等待结束
// 1位  gwaiting  --等待队列不为空，有g处于waiting状态
//    [15 bits] mcpu        number of m's executing on cpu
//    [15 bits] mcpumax    max number of m's allowed on cpu
//    [1 bit] waitstop    some g is waiting on stopped
//    [1 bit] gwaiting    gwait != 0
```

这些信息是进行系统调用和出系统调用时需要用到的，它会决定是否需要进入到调度器层面。直接用CAS操作Sched的atomic字段判断，将它们打包成一个字节使得可以通过一次原子读写获取它们而不用加锁。这将极大的减少那些大量使用系统调用或者cgo的多线程程序的contention。

除了进出系统调用以外，操作这些域只会发生于持有调度器锁的时候，因此goroutines不用担心其它goroutine会对这些字段进行操作。特别是，进出系统调用只会读mcpumax，waitstop和gwaiting。决不会写他们。因此，(持有调度器锁)写这些域时完全不用担心会发生写冲突。

总体上看，Go1.0调度设计结构比较简单，代码也比较清晰。但是也存在一些问题。这样的调度器设计限制了Go程序的并发度。测试发现有14%是的时间浪费在了runtime.futex()中。

具体地看：

1. 单个全局锁(Sched.Lock)用来保护所有的goroutine相关的操作(创建，完成，调度等)。
2. Goroutine切换。工作线程在各自之前切换goroutine，这导致延迟和额外的负担。每个M都必须可以执行任何的G.
3. 内存缓存MCache是每个M的。而当M阻塞后，相应的内存资源也被一起拿走了。
4. 过多的线程阻塞、恢复。系统调用时的工作线程会频繁地阻塞，恢复，造成过多的负担。

第一点很明显，所有的goroutine都用一个锁保护的，这个锁粒度是比较大的，只要goroutine的相关操作都会锁住调度。然后是goroutine切换，前面说了，每个M都是可以执行所有的goroutine的。举个很简单的类比，多核CPU中每个核都去执行不同线程的代码，这显然是不利于缓存的局部性的，切换开销也会变大。内存缓存和其它缓存是关联到所有的M的，而事实上它本只需要关联到运行Go代码的M(阻塞于系统调用的M是不需要mcache的)。运行着Go代码的M和所有M的比例可能高达1:100。这导致过度的资源消耗。

Go1.1

Go1.1相对于1.0一个重要的改动就是重新调用了调度器。前面已经看到，老版本中的调度器实现是存在一些问题的。解决方式是引入Processor的概念，并在Processors之上实现工作流窃取的调度器。

M代表OS线程。P代表Go代码执行时需要的资源。当M执行Go代码时，它需要关联一个P，当M为idle或者在系统调用中时，它也需要P。有刚好GOMAXPROCS个P。所有的P被组织为一个数组，工作流窃取需要这个条件。GOMAXPROCS的改变涉及到stop/start the world来resize数组P的大小。

gfree和grunnable从sched中移到P中。这样就解决了前面的单个全局锁保护用有goroutine的问题，由于goroutine现在被分到每个P中，它们是P局部的goroutine，因此P只管去操作自己的goroutine就行了，不会与其它P上的goroutine冲突。全局的grunnable队列也仍然是存在的，只有在P去访问全局grunnable队列时才涉及到加锁操作。mcache从M中移到P中。不过当前还不彻底，在M中还是保留着mcache域的。

加入了P后，sched.atomic也从Sched结构体中去掉了。

当一个新的G创建或者现有的G变成runnable，它将一个runnable的goroutine推到当前的P。当P完成执行G，它将G从自己的runnable goroutine中pop出去。如果链为空，P会随机从其它P中窃取一半的可运行的goroutine。

当M创建一个新G的时候，必须保证有另一个M来执行这个G。类似的，当一个M进入到系统调用时，必须保证有另一个M来执行G的代码。

2层自旋：关联了P的处于idle状态的的M自旋寻找新的G；没有关联P的M自旋等待可用的P。最多有GOMAXPROCS个自旋的M。只要有第二类M时第一类M就不会阻塞。

5.5 抢占式调度

goroutine本来是设计为协程形式，但是随着调度器的实现越来越成熟，Go在1.2版中开始引入比较初级的抢占式调度。

从一个bug说起

Go在设计之初并没考虑将goroutine设计成抢占式的。用户负责让各个goroutine交互合作完成任务。一个goroutine只有在涉及到加锁，读写通道或者主动让出CPU等操作时才会触发切换。

垃圾回收器是需要stop the world的。如果垃圾回收器想要运行了，那么它必须先通知其它的goroutine合作停下来，这会造成较长时间的等待时间。考虑一种很极端的情况，所有的goroutine都停下来了，只有其中一个没有停，那么垃圾回收就会一直等待着没有停的那一个。

抢占式调度可以解决这种问题，在抢占式情况下，如果一个goroutine运行时间过长，它就会被剥夺运行权。

总体思路

引入抢占式调度，会对最初的设计产生比较大的影响，Go还只是引入了一些很初级的抢占，并没有像操作系统调度那么复杂，没有对goroutine分时间片，设置优先级等。

只有长时间阻塞于系统调用，或者运行了较长时间才会被抢占。runtime会在后台有一个检测线程，它会检测这些情况，并通知goroutine执行调度。

目前并没有直接在后台的检测线程中做处理调度器相关逻辑，只是相当于给goroutine加了一个“标记”，然后在它进入函数时才会触发调度。这么做应该是出于对现有代码的修改最小的考虑。

sysmon

前面讲Go程序的初始化过程中有提到过，runtime开了一条后台线程，运行一个sysmon函数。这个函数会周期性地做epoll操作，同时它还会检测每个P是否运行了较长时间。

如果检测到某个P状态处于Psyscall超过了一个sysmon的时间周期(20us)，并且还有其它可运行的任务，则切换P。

如果检测到某个P的状态为Prunning，并且它已经运行了超过10ms，则会将P的当前的G的stackguard设置为StackPreempt。这个操作其实是相当于加上一个标记，通知这个G在合适时机进行调度。

目前这里只是尽最大努力送达，但并不保证收到消息的goroutine一定会执行调度让出运行权。

morestack的修改

前面说的，将stackguard设置为StackPreempt实际上是一个比较trick的代码。我们知道Go会在每个函数入口处比较当前的栈寄存器值和stackguard值来决定是否触发morestack函数。

将stackguard设置为StackPreempt作用是进入函数时必定触发morestack，然后在morestack中再引发调度。

看一下StackPreempt的定义，它是大于任何实际的栈寄存器的值的：

```
// 0xfffffade in hex.
#define StackPreempt ((uint64)-1314)
```

然后在morestack中加了一小段代码，如果发现stackguard为StackPreempt，则相当于调用runtime.Gosched。

所以，到目前为止Go的抢占式调度还是很初级的，比如一个goroutine运行了很久，但是它并没有调用另一个函数，则它不会被抢占。当然，一个运行很久却不调用函数的代码并不是多数情况。

6 内存管理

内存管理是非常重要的一个话题。关于编程语言是否应该支持垃圾回收就有个搞笑的争论，一派人认为，内存管理太重要了，而手动管理麻烦且容易出错，所以我们应该交给机器去管理。另一派人则认为，内存管理太重要了！所以如果交给机器管理我不能放心。争论归争论，但不管哪一派，大家对内存管理重要性的认同都是勿庸质疑的。

Go是一门带垃圾回收的语言，Go语言中有指针，却没有C中那么灵活的指针操作。大多数情况下是不需要用户自己去管理内存的，但是理解Go语言是如何做内存管理对于写出优秀的程序是大有帮助的。

本章将从两个方面来看Go中的内存管理机制，一个方面是内存池，另一个方面是垃圾回收。

links

- 目录
- 上一章: 抢占式调度
- 下一节: 内存池

6.1 内存池

概述

Go的内存分配器采用了跟tcmalloc库相同的实现，是一个带内存池的分配器，底层直接调用操作系统的mmap等函数。

作为一个内存池，回忆一下跟它相关的基本部分。首先，它会向操作系统申请大块内存，自己管理这部分内存。然后，它是一个池子，当上层释放内存时它不实际归还给操作系统，而是放回池子重复利用。接着，内存管理中必然会考虑的就是内存碎片问题，如果尽量避免内存碎片，提高内存利用率，像操作系统中的首次适应，最佳适应，最差适应，伙伴算法都是一些相关的背景知识。另外，Go是一个支持goroutine这种多线程的语言，所以它的内存管理系统必须也要考虑在多线程下的稳定性和效率问题。

在多线程方面，很自然的做法就是每条线程都有自己的本地的内存，然后有一个全局的分配链，当某个线程中内存不足后就向全局分配链中申请内存。这样就避免了多线程同时访问共享变量时的加锁。 在避免内存碎片方面，大块内存直接按页为单位分配，小块内存会切成各种不同的固定大小的块，申请做任意字节内存时会向上取整到最接近的块，将整块分配给申请者以避免随意切割。

Go中为每个系统线程分配一个本地的MCache(前面介绍的结构体M中的MCache域)，少量的地址分配就直接从MCache中分配，并且定期做垃圾回收，将线程的MCache中的空闲内存返回给全局控制堆。小于32K为小对象，大对象直接从全局控制堆上以页(4k)为单位进行分配，也就是说大对象总是以页对齐的。一个页可以存入一些相同大小的小对象，小对象从本地内存链表中分配，大对象从中心内存堆中分配。

大约有100种内存块类别，每一类别都有自己对象的空闲链表。小于32kB的内存分配被向上取整到对应的尺寸类别，从相应的空闲链表中分配。一页内存只可以被分裂成同一种尺寸类别的对象，然后由空闲链表分配器管理。

分配器的数据结构包括:

- FixAlloc: 固定大小(128kB)的对象的空闲链分配器,被分配器用于管理存储
- MHeap: 分配堆,按页的粒度进行管理(4kB)
- MSpan: 一些由MHeap管理的页
- MCentral: 对于给定尺寸类别的共享的free list
- MCache: 用于小对象的每M一个的cache

我们可以将Go语言的内存管理看成一个两级的内存管理结构，MHeap和MCache。上面一级管理的基本单位是页，用于分配大对象，每次分配都是若干连续的页，也就是若干个4KB的大小。使用的数据结构是MHeap和MSpan，用BestFit算法做分配，用位示图做回收。下面一级管理的基本单位是不同类型的固定大小的对象，更像一个对象池而不是内存池，用引用计数做回收。下面这一级使用的数据结构是MCache。

MHeap

MHeap层次用于直接分配较大(>32kB)的内存空间，以及给MCentral和MCache等下层提供空间。它管理的基本单位是MSpan。MSpan是一个表示若干连续内存页的数据结构，简化后如下：

```
struct MSpan
{
    PageID    start;        // starting page number
    uintptr    npages;        // number of pages in span
};
```

通过一个基地址+(页号*页大小)，就可以定位到这个MSpan的实际的地址空间了，基地址是在MHeap中存储了的。

MHeap负责将MSpan组织和管理起来，MHeap数据结构中的重要部分如图所示。

free是一个分配池，从free[i]出去的MSpan每个大小都i页的,总共256个槽位。再大了之后，大小就不固定了，由large链起来。

分配过程： 如果能从free[]的分配池中分配，则从其中分配。如果发生切割则将剩余部分放回free[]中。比如要分配2页大小的空间，从图上2号槽位开始寻找，直到4号槽位有可用的MSpan，则拿一个出来，切出两页，剩余的部分再放回2号槽位中。否则从large链表中去分配，按BestFit算法去找一块可用空间。

化整为零简单，化零为整麻烦。回收的时候如果相邻的块是未使用的，要进行合并，否则一直划分下去就会产生很多碎片，找不到一个足够大小的连续空间。因为涉及到合并，回收会比分配复杂一些，所有就有什么伙伴算法，边界标识算法，位示图之类的。

Go在这里使用的类似于位示图。可以看到MHeap中有一个

```
MSpan *map[1<<MHeapMap_Bits];
```

这个数组是一个用于将内存地址映射成MSpan结构体的表，每个内存页都会对应到map中的一个MSpan指针，通过map就能够将地址映射到相应的MSpan。具体做法，给定一个地址，可以通过(地址-基地址)/页大小得到页号，再通过map[页号]就得到了相应的MSpan结构体。前面说过，MSpan就是若干连续的页。那么，一个多页的MSpan会占用map数组中的多项，有多少页就会占用多少项。比如，可能map[502]到map[505]都指向同一个MSpan，这个MSpan的PageId为502，npages为4。

回收过程： 回收一个MSpan时，首先会查找它相邻的页的址址，再通过map映射得到该页对应的MSpan，如果MSpan的state是未使用，则可以将两者进行合并。最后会将这页或者合并后的页归还到free[]分配池或者是large中。

MCache

MCache层次跟MHeap层次非常像，也是一个分配池，对每个尺寸的类别都有一个空闲对象的单链表。Go的内存管理可以看成一个两级的层次，上面一级是MHeap层次，而MCache则是下面一级。

每个M都有一个自己的局部内存缓存MCache，这样分配小对象的时候直接从MCache中分配，就不用加锁了，这是Go能够在多线程环境中高效地进行内存分配的重要原因。MCache是用于小对象的分配。

分配一个小对象(<32kB)的过程：

1. 将小对象大小向上取整到一个对应的尺寸类别，查找相应的MCache的空闲链表。如果链表不空，直接从上面分配一个对象。这个过程可以不必加锁。
2. 如果MCache自由链是空的,通过从MCentral自由链拿一些对象进行补充。
3. 如果MCentral自由链是空的,则通过MHeap中拿一些页对MCentral进行补充，然后将这些内存截断成规定的大小。
4. 如果MHeap是空的,或者没有足够大小的页了,从操作系统分配一组新的页(至少1MB)。分配一大批的页分摊了从操作系统分配的开销。

注意上面表述中的用词“一些”。从MCentral中拿“一些“自由链对象补充MCache分摊了访问MCentral加锁的开销。从MHeap中分配“一些“的页补充MCentral分摊了对MHeap加锁的开销。

释放一个小对象也是类似的过程:

1. 查找对象所属的尺寸类别，将它添加到MCache的自由链。
2. 如果MCache自由链太长或者MCache内存太多了，则返还一些到MCentral自由链。
3. 如果在某个范围的所有的对象都归还到MCentral链了，则将它们归还到页堆。

归还到MHeap就结束了，目前还是没有归还到操作系统。

MCache层次仅用于分配小对象，分配和释放大的对象则是直接使用MHeap的，跳过MCache和MCentral自由链。MCache和MCentral中自由链的小对象可能是也可能不是清0了的。对象的第2个字节作为标记，当它是0时，此对象是清0了的。页堆中的总是清零的，当一定范围的对象归还到页堆时，需要先清零。这样才符合Go语言规范：分配一个对象不进行初始化，它的默认值是该类型的零值。

MCentral

MCentral层次是作为MCache和MHeap的连接。对上，它从MHeap中申请MSpan；对下，它将MSpan划分成各种小尺寸对象，提供给MCache使用。

```
    struct MCentral
    {
        Lock;
        int32 sizeclass;
        MSpan nonempty;
        MSpan empty;
        int32 nfree;
    };
```

注意，每个MSpan只会分割成同种大小的对象。每个MCentral也是只含同种大小的对象。MCentral结构中，有一个nonempty的MSpan链和一个empty的MSpan链，分别表示还有空间的MSpan和装满了对象的MSpan。如图所示：

分配还是很简单，直接从MCentral->nonempty->freelist分配。如果发现freelist空了，则说明这一块MSpan满了，将它移到MCentral->empty。 前面说过，回收比分配复杂，因为涉及到合并。这里的合并是通过引用计数实现的。从MSpan中每划出一个对象，则引用计数加一，每回收一个对象，则引用计数减一。如果减完之后引用计数为零了，则说明这整块的MSpan已经没被使用了，可以将它归还给MHeap。

其它

本节的内存池涉及的文件包括：

- malloc.h 头文件
- malloc.goc 最外层的包装
- msize.c 将各种大小向上取整到相应的尺寸类别
- mheap.c 对应MHeap中相关实现,还有MSpan
- mcache.c 对应MCache中相关实现
- mcentral.c 对应MCentral中相关实现
- mem_linux.c SysAlloc等sys相关的实现

6.2 垃圾回收

Go语言中使用的垃圾回收使用的是标记清扫算法。进行垃圾回收时会stoptheworld。不过，在当前1.3版本中，实现了精确的垃圾回收和并行的垃圾回收，大大地提高了垃圾回收的速度，进行垃圾回收时系统并不会长时间卡住。

标记清扫算法

标记清扫算法是一个很基础的垃圾回收算法，该算法中有一个标记初始的root区域，以及一个受控堆区。root区域主要是程序运行到当前时刻的栈和全局数据区域。在受控堆区中，很多数据是程序以后不需要用到的，这类数据就可以被当作垃圾回收了。判断一个对象是否为垃圾，就是看从root区域的对象是否有直接或间接的引用到这个对象。如果没有任何对象引用到它，则说明它没有被使用，因此可以安全地当作垃圾回收掉。

标记清扫算法分为两阶段：标记阶段和清扫阶段。标记阶段，从root区域出发，扫描所有root区域的对象直接或间接引用到的对象，将这些对上全部加上标记。在回收阶段，扫描整个堆区，对所有无标记的对象进行回收。(补图)

位图标记和内存布局

既然垃圾回收算法要求给对象加上垃圾回收的标记，显然是需要有标记位的。一般的做法会将对象结构体中加上一个标记域，一些优化的做法会利用对象指针的低位进行标记，这都只是些奇技淫巧罢了。Go没有这么做，它的对象和C的结构体对象完全一致，使用的是非侵入式的标记位，我们看看它是怎么实现的。

堆区域对应了一个标记位图区域，堆中每个字(不是byte，而是word)都会在标记位区域中有对应的标记位。每个机器字(32位或64位)会对应4位的标记位。因此，64位系统中相当于每个标记位图的字节对应16个堆中的字节。

虽然是一个堆字节对应4位标记位，但标记位图区域的内存布局并不是按4位一组，而是16个堆字节为一组，将它们的标记位信息打包存储的。每组64位的标记位图从上到下依次包括：

```
16位的 特殊位 标记位
16位的 垃圾回收 标记位
16位的 无指针/块边界 的标记位
16位的 已分配 标记位
```

这样设计使得对一个类型的相应的位进行遍历很容易。

前面提到堆区域和堆地址的标记位图区域是分开存储的，其实它们是以mheap.arena_start地址为边界，向上是实际使用的堆地址空间，向下则是标记位图区域。以64位系统为例，计算堆中某个地址的标记位的公式如下：

```
偏移 = 地址 - mheap.arena_start
标记位地址 = mheap.arena_start - 偏移/16 - 1
移位 = 偏移 % 16
标记位 = *标记位地址 >> 移位
```

然后就可以通过 (标记位 & 垃圾回收标记位),(标记位 & 分配位),等来测试相应的位。其中已分配的标记为1<<0,无指针/块边界是1<<16,垃圾回收的标记位为1<<32,特殊位1<<48

具体的内存布局如下图所示:

精确的垃圾回收

像C这种不支持垃圾回收的语言，其实还是有些垃圾回收的库可以使用的。这类库一般也是用的标记清扫算法实现的，但是它们都是保守的垃圾回收。为什么叫“保守”的垃圾回收呢？之所以叫“保守”是因为它们没办法获取对象类型信息，因此只能保守地假设地址区间中每个字都是指针。

无法获取对象的类型信息会造成什么问题呢？这里举两个例子来说明。先看第一个例子，假设某个结构体中是不包含指针成员的，那么对该结构体成员进行垃圾回收时，其实是不必要递归地标记结构体的成员的。但是由于没有类型信息，我们并不知道这个结构体成员不包含指针，因此我们只能对结构体的每个字节递归地标记下去，这显然会浪费很多时间。这个例子说明精确的垃圾回收可以减少不必要的扫描，提高标记过程的速度。

再看另一个例子，假设堆中有一个long的变量，它的值是8860225560。但是我们不知道它的类型是long，所以在进行垃圾回收时会把个当作指针处理，这个指针引用到了0x2101c5018位置。假设0x2101c5018碰巧有某个对象，那么这个对象就无法被释放了，即使实际上已经没任何地方使用它。这个例子说明，保守的垃圾回收某些情况下会出现垃圾无法被回收。虽然不会造成大的问题，但总是让人很不爽，都是没有类型信息惹的祸。

现在好了，Go在1.1版本中开始支持精确的垃圾回收。精确的垃圾回收首先需要的就是类型信息，上一节中讲过MSpan结构体，类型信息是存储在MSpan中的。从一个地址计算它所属的MSpan，公式如下：

```
页号 = (地址 - mheap.arena_start) >> 页大小
MSpan = mheap->map[页号]
```

接下来通过MSpan->type可以得到分配块的类型。这是一个MType的结构体：

```
    struct MTypes
    {
        byte    compression;    // one of MTypes_*
        bool    sysalloc;    // whether (void*)data is from runtime·SysAlloc
        uintptr    data;
    };
```

MTypes描述MSpan里分配的块的类型，其中compression域描述数据的布局。它的取值为MTypes_Empty，MTypes_Single，MTypes_Words，MTypes_Bytes四个中的一种。

```
MTypes_Empty:
    所有的块都是free的，或者这个分配块的类型信息不可用。这种情况下data域是无意义的。
MTypes_Single:
    这个MSpan只包含一个块，data域存放类型信息，sysalloc域无意义
MTypes_Words:
    这个MSpan包含多个块(块的种类多于7)。这时data指向一个数组[NumBlocks]uintptr,，数组里每个元素存放相应块的类型信息
MTypes_Bytes:
    这个MSpan中包含最多7种不同类型的块。这时data域指下面这个结构体
    struct {
        type  [8]uintptr       // type[0] is always 0
        index [NumBlocks]byte
    }
    第i个块的类型是data.type[data.index[i]]
```

表面上看MTypes_Bytes好像最复杂，其实这里的复杂程度是MTypes_Empty小于MTypes_Single小于MTypes_Bytes小于MTypes_Words的。MTypes_Bytes只不过为了做优化而显得很复杂。

上一节中说过，每一块MSpan中存放的块的大小都是一样的，不过它们的类型不一定相同。如果没有使用，那么这个MSpan的类型就是MTypes_Empty。如果存一个很大块，大于这个MSpan大小的一半，因此存不了其它东西了，那么这个MSpan的类型是MTypes_Single。假设存了多种块，每一块用一个指针，本来可以直接用MTypes_Words存的。但是当类型不多时，可以把这些类型的指针集中起来放在数组中，然后存储数组索引。这是一个小的优化，可以节省内存空间。

得到的类型信息最终是什么样子的呢？其实是一个这样的结构体：

```
struct Type
{
    uintptr size;
    uint32 hash;
    uint8 _unused;
    uint8 align;
    uint8 fieldAlign;
    uint8 kind;
    Alg *alg;
    void *gc;
    String *string;
    UncommonType *x;
    Type *ptrto;
};
```

不同类型的类型信息结构体略有不同，这个是通用的部分。可以看到这个结构体中有一个gc域，精确的垃圾回收就是利用类型信息中这个gc域实现的。

从gc出去其实是一段指令码，是对这种类型的数据进行垃圾回收的指令，Go中用一个状态机来执行垃圾回收指令码。大致的框架是类似下面这样子：

```
    for(;;) {
        switch(pc[0]) {
            case GC_PTR:
            break;
            case GC_SLICE:
            break;
            case GC_APTR:
            break;
            case GC_STRING:
            continue;
            case GC_EFACE:
            if(eface->type == nil)
                continue;
            break;
            case GC_IFACE:
            break;
            case GC_DEFAULT_PTR:
            while(stack_top.b <= end_b) {
                obj = *(byte**)stack_top.b;
                stack_top.b += PtrSize;
                if(obj >= arena_start && obj < arena_used) {
                    *ptrbufpos++ = (PtrTarget){obj, 0};
                    if(ptrbufpos == ptrbuf_end)
                        flushptrbuf(ptrbuf, &ptrbufpos, &wp, &wbuf, &nobj);
                }
            }
            case GC_ARRAY_START:
            continue;
            case GC_ARRAY_NEXT:
            continue;
            case GC_CALL:
            continue;
            case GC_MAP_PTR:
            continue;
            case GC_MAP_NEXT:
            continue;
            case GC_REGION:
            continue;
            case GC_CHAN_PTR:
            continue;
            case GC_CHAN:
            continue;
            default:
            runtime·throw("scanblock: invalid GC instruction");
            return;
        }
    }
```

小结

Go语言使用标记清扫的垃圾回收算法，标记位图是非侵入式的，内存布局设计得比较巧妙。并且当前版本的Go实现了精确的垃圾回收。在精确的垃圾回收中，通过定位对象的类型信息，得到该类型中的垃圾回收的指令码，通过一个状态机解释这段指令码来执行特定类型的垃圾回收工作。

对于堆中任意地址的对象，找到它的类型信息过程为，先通过它在的内存页找到它所属的MSpan，然后通过MSpan中的类型信息找到它的类型信息。

不知道读者有没有注意一个细节，MType中的data值应该是存放Type结构体的指针，但它却是uintptr表示的。这是为什么呢？

6.3 垃圾回收

目前Go中垃圾回收的核心函数是scanblock，源代码在文件runtime/mgc0.c中。这个函数非常难读，单个函数写了足足500多行。上面有两个大的循环，外层循环作用是扫描整个内存块区域，将类型信息提取出来，得到其中的gc域。内层的大循环是实现一个状态机，解析执行类型信息中gc域的指令码。

先说说上一节留的疑问吧。MType中的数据其实是类型信息，但它是用uintptr表示，而不是Type结构体的指针，这是一个优化的小技巧。由于内存分配是机器字节对齐的，所以地址就只用到了高位，低位是用不到的。于是低位可以利用起来存储一些额外的信息。这里的uintptr中高位存放的是Type结构体的指针，低位用来存放类型。通过

```
    t = (Type*)(type & ~(uintptr)(PtrSize-1));
```

就可以从uintptr得到Type结构体指针，而通过

```
type & (PtrSize-1)
```

就可以得到类型。这里的类型有TypeInfo_SingleObject，TypeInfo_Array，TypeInfo_Map，TypeInfo_Chan几种。

基本的标记过程

从最简单的开始看，基本的标记过程，有一个不带任何优化的标记的实现，对应于函数debug_scanblock。

debug_scanblock函数是递归实现的，单线程的，更简单更慢的scanblock版本。该函数接收的参数分别是一个指针表示要扫描的地址，以及字节数。

```
首先要将传入的地址，按机器字节大小对齐。
然后对待扫描区域的每个地址：
找到它所属的MSpan，将地址转换为MSpan里的对象地址。
根据对象的地址，找到对应的标记位图里的标记位。
判断标记位，如果是未分配则跳过。否则加上特殊位标记(debug_scanblock中用特殊位代码的mark位)完成标记。
判断标记位中标记了无指针标记位，如果没有，则要递归地调用debug_scanblock。
```

这个递归版本的标记算法还是很容易理解的。其中涉及的细节在上节中已经说过了，比如任意给定一个地址，找到它的标记位信息。很明显这里仅仅使用了一个无指针位，并没有精确的垃圾回收。

并行的垃圾回收

Go在这个版本中不仅实现了精确的垃圾回收，而且实现了并行的垃圾回收。标记算法本质上就是一个树的遍历过程，上面实现的是一个递归版本。

并行的垃圾回收需要做的第一步，就是先将算法做成非递归的。非递归版本的树的遍历需要用到一个队列。树的非递归遍历的伪代码大致是：

```
根结点进队
while(队列不空) {
    出队
    访问
    将子结点进队
}
```

第二步是使上面的代码能够并行地工作，显然这时是需要一个线程安全的队列的。假设有这样一个队列，那么上面代码就能够工作了。但是，如果不加任何优化，这里的队列的并行访问非常地频繁，对这个队列加锁代价会非常高，即使是使用CAS操作也会大大降低效率。

所以，第三步要做的就是优化上面队列的数据结构。事实上，Go中并没有使用这样一个队列，为了优化，它通过三个数据结构共同来完成这个队列的功能，这三个数据结构分别是PtrTarget数组，Workbuf，lfstack。

先说Workbuf吧。听名字就知道，这个结构体的意思是工作缓冲区，里面存放的是一个数组，数组中的每个元素都是一个待处理的结点，也就是一个Obj指针。这个对象本身是已经标记了的，这个对象直接或间接引用到的对象，都是应该被标记的，它们不会被当作垃圾回收掉。Workbuf是比较大的，一般是N个内存页的大小(目前是2页，也就是8K)。

PtrTarget数组也是一个缓冲区，相当于一个intermediate buffer，跟Workbuf有一点点的区别。第一，它比Workbuf小很多，大概只有32或64个元素的数组。第二，Workbuf中的对象全部是已经标记过的，而PtrTarget中的元素可能是标记的，也可能是没标记的。第三，PtrTarget里面的元素是指针而不是对象，指针是指向任意地址的，而对象是对齐到正确地址的。从一个指针变为一个对象要经过一次变换，上一节中有讲过具体细节。

垃圾回收过程中，会有一个从PtrTarget数组冲刷到Workbuf缓冲区的过程。对应于源代码中的flushptrbuf函数，这个函数作用就是对PtrTaget数组中的所有元素，如果该地址是mark了的，则将它移到Workbuf中。标记过程形成了一个环，在环的一边，对Workbuf中的对象，会将它们可能引用的区域全部放到PtrTarget中记录下来。在环的另一边，又会将PtrTarget中确定需要标记的地址刷到Workbuf中。这个过程一轮一轮地进行，推动非递归版本的树的遍历过程，也就是前面伪代码中的出队，访问，子结点进队的过程。

另一个数据结构是lfstack，这个名字的意思是lock free栈。其实它是被用作了一个无锁的链表，链表结点是以Workbuf为单位的。并行垃圾回收中，多条线程会从这个链表中取数据，每次以一个Workbuf为工作单位。同时，标记的过程中也会产生Workbuf结点放到链中。lfstack保证了对这个链的并发访问的安全性。由于现在链表结点是以Workbuf为单位的，所以保证整体的性能，lfstack的底层代码是用CAS操作实现的。

经过第三步中数据结构上的拆解，整个并行垃圾回收的架构已经呼之欲出了，这就是标记扫描的核心函数scanblock。这个函数是在多线程下并行安全的。

那么，最后一步，多线程并行。整个的gc是以runtime.gc函数为入口的，它实际调用的是gc。进入gc函数后会先stoptheworld，接着添加标记的root区域。然后会设置markroot和sweepspan的并行任务。运行mark的任务，扫描块，运行sweep的任务，最后starttheworld并切换出去。

有一个ParFor的数据结构。在gc函数中调用了

```
runtime·parforsetup(work.markfor, work.nproc, work.nroot, nil, false, markroot);
runtime·parforsetup(work.sweepfor, work.nproc, runtime·mheap->nspan, nil, true, sweepspan);
```

是设置好回调函数让线程去执行markroot和sweepspan函数。垃圾回收时会stoptheworld，其它goroutine会对发起stoptheworld做出响应，调用runtime.gchelper，这个函数会调用scanblock帮助标记过程。也会并行地做markroot和sweepspan的过程。

```
void
runtime·gchelper(void)
{
    gchelperstart();

    // parallel mark for over gc roots
    runtime·parfordo(work.markfor);

    // help other threads scan secondary blocks
    scanblock(nil, nil, 0, true);

    if(DebugMark) {
        // wait while the main thread executes mark(debug_scanblock)
        while(runtime·atomicload(&work.debugmarkdone) == 0)
            runtime·usleep(10);
    }

    runtime·parfordo(work.sweepfor);
    bufferList[m->helpgc].busy = 0;
    if(runtime·xadd(&work.ndone, +1) == work.nproc-1)
        runtime·notewakeup(&work.alldone);
}
```

其中并行时也有实现工作流窃取的概念，多个worker同时去工作缓存中取数据出来处理，如果自己的任务做完了，就会从其它的任务中“偷”一些过来执行。

垃圾回收的时机

垃圾回收的触发是由一个gcpercent的变量控制的，当新分配的内存占已在使用中的内存的比例超过gcprecent时就会触发。比如，gcpercent=100，当前使用了4M的内存，那么当内存分配到达8M时就会再次gc。如果回收完毕后，内存的使用量为5M，那么下次回收的时机则是内存分配达到10M的时候。也就是说，并不是内存分配越多，垃圾回收频率越高，这个算法使得垃圾回收的频率比较稳定，适合应用的场景。

gcpercent的值是通过环境变量GOGC获取的，如果不设置这个环境变量，默认值是100。如果将它设置成off，则是关闭垃圾回收。

7.1 channel

channel数据结构

Go语言channel是first-class的，意味着它可以被存储到变量中，可以作为参数传递给函数，也可以作为函数的返回值返回。作为Go语言的核心特征之一，虽然channel看上去很高端，但是其实channel仅仅就是一个数据结构而已，结构体定义如下：

```
struct    Hchan
{
    uintgo    qcount;            // 队列q中的总数据数量
    uintgo    dataqsiz;        // 环形队列q的数据大小
    uint16    elemsize;
    bool    closed;
    uint8    elemalign;
    Alg*    elemalg;        // interface for element type
    uintgo    sendx;            // 发送index
    uintgo    recvx;            // 接收index
    WaitQ    recvq;            // 因recv而阻塞的等待队列
    WaitQ    sendq;            // 因send而阻塞的等待队列
    Lock;
};
```

让我们来看一个Hchan这个结构体。其中一个核心的部分是存放channel数据的环形队列，由qcount和elemsize分别指定了队列的容量和当前使用量。dataqsize是队列的大小。elemalg是元素操作的一个Alg结构体，记录下元素的操作，如copy函数，equal函数，hash函数等。

可能会有人疑惑，结构体中只看到了队列大小相关的域，并没有看到存放数据的域啊？如果是带缓冲区的chan，则缓冲区数据实际上是紧接着Hchan结构体中分配的。

```
c = (Hchan*)runtime.mal(n + hint*elem->size);
```

另一个重要部分就是recvq和sendq两个链表，一个是因读这个通道而导致阻塞的goroutine，另一个是因为写这个通道而阻塞的goroutine。如果一个goroutine阻塞于channel了，那么它就被挂在recvq或sendq中。WaitQ是链表的定义，包含一个头结点和一个尾结点：

```
struct    WaitQ
{
    SudoG*    first;
    SudoG*    last;
};
```

队列中的每个成员是一个SudoG结构体变量。

```
struct    SudoG
{
    G*    g;        // g and selgen constitute
    uint32    selgen;        // a weak pointer to g
    SudoG*    link;
    int64    releasetime;
    byte*    elem;        // data element
};
```

该结构中主要的就是一个g和一个elem。elem用于存储goroutine的数据。读通道时，数据会从Hchan的队列中拷贝到SudoG的elem域。写通道时，数据则是由SudoG的elem域拷贝到Hchan的队列中。

Hchan结构如下图所示:

读写channel操作

先看写channel的操作，基本的写channel操作，在底层运行时库中对应的是一个runtime.chansend函数。

```
c <- v
```

在运行时库中会执行：

```
void runtime·chansend(ChanType *t, Hchan *c, byte *ep, bool *pres, void *pc)
```

其中c就是channel，ep是取变量v的地址。这里的传值约定是调用者负责分配好ep的空间，仅需要简单的取变量地址就够了。pres参数是在select中的通道操作使用的。

这个函数首先会区分是同步还是异步。同步是指chan是不带缓冲区的，因此可能写阻塞，而异步是指chan带缓冲区，只有缓冲区满才阻塞。

在同步的情况下，由于channel本身是不带数据缓存的，这时首先会查看Hchan结构体中的recvq链表时否为空，即是否有因为读该管道而阻塞的goroutine。如果有则可以正常写channel，否则操作会阻塞。

recvq不为空的情况下，将一个SudoG结构体出队列，将传给通道的数据(函数参数ep)拷贝到SudoG结构体中的elem域，并将SudoG中的g放到就绪队列中，状态置为ready，然后函数返回。

如果recvq为空，否则要将当前goroutine阻塞。此时将一个SudoG结构体，挂到通道的sendq链表中，这个SudoG中的elem域是参数eq，SudoG中的g是当前的goroutine。当前goroutine会被设置为waiting状态并挂到等待队列中。

在异步的情况，如果缓冲区满了，也是要将当前goroutine和数据一起作为SudoG结构体挂在sendq队列中，表示因写channel而阻塞。否则也是先看有没有recvq链表是否为空，有就唤醒。

跟同步不同的是在channel缓冲区不满的情况，这里不会阻塞写者，而是将数据放到channel的缓冲区中，调用者返回。

读channel的操作也是类似的，对应的函数是runtime.chansend。一个是收一个是发，基本的过程都是差不多的。

需要注意的是几种特殊情况下的通道操作--空通道和关闭的通道。

空通道是指将一个channel赋值为nil，或者定义后不调用make进行初始化。按照Go语言的语言规范，读写空通道是永远阻塞的。其实在函数runtime.chansend和runtime.chanrecv开头就有判断这类情况，如果发现参数c是空的，则直接将当前的goroutine放到等待队列，状态设置为waiting。

读一个关闭的通道，永远不会阻塞，会返回一个通道数据类型的零值。这个实现也很简单，将零值复制到调用函数的参数ep中。写一个关闭的通道，则会panic。关闭一个空通道，也会导致panic。

select的实现

select-case中的chan操作编译成了if-else。比如：

```
select {
case v = <-c:
        ...foo
default:
        ...bar
}
```

会被编译为:

```
if selectnbrecv(&v, c) {
        ...foo
} else {
        ...bar
}
```

类似地

```
select {
case v, ok = <-c:
    ... foo
default:
    ... bar
}
```

会被编译为:

```
if c != nil && selectnbrecv2(&v, &ok, c) {
    ... foo
} else {
    ... bar
}
```

接下来就是看一下selectnbrecv相关的函数了。其实没有任何特殊的魔法，这些函数只是简单地调用runtime.chanrecv函数，只不过设置了一个参数，告诉当runtime.chanrecv函数，当不能完成操作时不要阻塞，而是返回失败。也就是说，所有的select操作其实都仅仅是被换成了if-else判断，底层调用的不阻塞的通道操作函数。

在Go的语言规范中，select中的case的执行顺序是随机的，而不像switch中的case那样一条一条的顺序执行。那么，如何实现随机呢？

select和case关键字使用了下面的结构体：

```
struct    Scase
{
    SudoG    sg;            // must be first member (cast to Scase)
    Hchan*    chan;        // chan
    byte*    pc;            // return pc
    uint16    kind;
    uint16    so;            // vararg of selected bool
    bool*    receivedp;    // pointer to received bool (recv2)
};

struct    Select
{
    uint16    tcase;            // 总的scase[]数量
    uint16    ncase;            // 当前填充了的scase[]数量
    uint16*    pollorder;        // case的poll次序
    Hchan**    lockorder;        // channel的锁住的次序
    Scase    scase[1];        // 每个case会在结构体里有一个Scase，顺序是按出现的次序
};
```

每个select都对应一个Select结构体。在Select数据结构中有个Scase数组，记录下了每一个case，而Scase中包含了Hchan。然后pollorder数组将元素随机排列，这样就可以将Scase乱序了。

7.2 interface

interface是Go语言中最成功的设计之一，空的interface可以被当作“鸭子”类型使用，它使得Go这样的静态语言拥有了一定的动态性，但却又不损失静态语言在类型安全方面拥有的编译时检查的优势。

依赖于接口而不是实现，优先使用组合而不是继承，这是程序抽象的基本原则。但是长久以来以C++为代表的“面向对象”语言曲解了这些原则，让人们走入了误区。为什么要将方法和数据绑死？为什么要有多重继承这么变态的设计？面向对象中最强调的应该是对象间的消息传递，却为什么被演绎成了封装继承和多态。面向对象是否实现程序程序抽象的合理途径，又或者是因为它存在我们就认为它合理了。历史原因，中间出现了太多的错误。不管怎么样，Go的interface给我们打开了一扇新的窗。

那么，Go中的interface在底层是如何实现的呢？

Eface和Iface

interface实际上就是一个结构体，包含两个成员。其中一个成员是指向具体数据的指针，另一个成员中包含了类型信息。空接口和带方法的接口略有不同，下面分别是空接口和带方法的接口是使用的数据结构：

```
struct Eface
{
    Type*    type;
    void*    data;
};
struct Iface
{
    Itab*    tab;
    void*    data;
};
```

先看Eface，它是interface{}底层使用的数据结构。数据域中包含了一个void*指针，和一个类型结构体的指针。interface{}扮演的角色跟C语言中的void*是差不多的，Go中的任何对象都可以表示为interface{}。不同之处在于，interface{}中有类型信息，于是可以实现反射。

类型信息的结构体定义如下：

```
struct Type
{
    uintptr size;
    uint32 hash;
    uint8 _unused;
    uint8 align;
    uint8 fieldAlign;
    uint8 kind;
    Alg *alg;
    void *gc;
    String *string;
    UncommonType *x;
    Type *ptrto;
};
```

其实在前面我们已经见过它了。精确的垃圾回收中，就是依赖Type结构体中的gc域的。不同类型数据的类型信息结构体并不完全一致，Type是类型信息结构体中公共的部分，其中size描述类型的大小，hash数据的hash值，align是对齐，fieldAlgin是这个数据嵌入结构体时的对齐，kind是一个枚举值，每种类型对应了一个编号。alg是一个函数指针的数组，存储了hash/equal/print/copy四个函数操作。UncommonType是指向一个函数指针的数组，收集了这个类型的实现的所有方法。

在reflect包中有个KindOf函数，返回一个interface{}的Type，其实该函数就是简单的取Eface中的Type域。

Iface和Eface略有不同，它是带方法的interface底层使用的数据结构。data域同样是指向原始数据的，而Itab的结构如下：

```
struct    Itab
{
    InterfaceType*    inter;
    Type*    type;
    Itab*    link;
    int32    bad;
    int32    unused;
    void    (*fun[])(void);
};
```

Itab中不仅存储了Type信息，而且还多了一个方法表fun[]。一个Iface中的具体类型中实现的方法会被拷贝到Itab的fun数组中。

具体类型向接口类型赋值

将具体类型数据赋值给interface{}这样的抽象类型，中间会涉及到类型转换操作。从接口类型转换为具体类型(也就是反射)，也涉及到了类型转换。这个转换过程中做了哪些操作呢？先看将具体类型转换为接口类型。如果是转换成空接口，这个过程比较简单，就是返回一个Eface，将Eface中的data指针指向原型数据，type指针会指向数据的Type结构体。

将某个类型数据转换为带方法的接口时，会复杂一些。中间涉及了一道检测，该类型必须要实现了接口中声明的所有方法才可以进行转换。这个检测是在编译过程中做的，我们可以做个测试：

```
type I interface {
    String()
}
var a int = 5
var b I = a
```

编译会报错：

```
cannot use a (type int) as type I in assignment:
    int does not implement I (missing String method)
```

说明具体类型转换为带方法的接口类型是在编译过程中进行检测的。

那么这个检测是如何实现的呢？在runtime下找到了iface.c文件，应该是早期版本是在运行时检测留下的，其中有一个itab函数就是判断某个类型是否实现了某个接口，如果是则返回一个Itab结构体。

类型转换时的检测就是比较具体类型的方法表和接口类型的方法表，看具体类型是实现了接口类型所声明的所有的方法。还记得Type结构体中是有个UncommonType字段的，里面有张方法表，类型所实现的方法都在里面。而在Itab中有个InterfaceType字段，这个字段中也有一张方法表，就是这个接口所要求的方法。这两处方法表都是排序过的，只需要一遍顺序扫描进行比较，应该可以知道Type中否实现了接口中声明的所有方法。最后还会将Type方法表中的函数指针，拷贝到Itab的fun字段中。

这里提到了三个方法表，有点容易把人搞晕，所以要解释一下。

Type的UncommonType中有一个方法表，某个具体类型实现的所有方法都会被收集到这张表中。reflect包中的Method和MethodByName方法都是通过查询这张表实现的。表中的每一项是一个Method，其数据结构如下：

```
struct Method
{
    String *name;
    String *pkgPath;
    Type    *mtyp;
    Type *typ;
    void (*ifn)(void);
    void (*tfn)(void);
};
```

Iface的Itab的InterfaceType中也有一张方法表，这张方法表中是接口所声明的方法。其中每一项是一个IMethod，数据结构如下：

```
struct IMethod
{
    String *name;
    String *pkgPath;
    Type *type;
};
```

跟上面的Method结构体对比可以发现，这里是只有声明没有实现的。

Iface中的Itab的func域也是一张方法表，这张表中的每一项就是一个函数指针，也就是只有实现没有声明。

类型转换时的检测就是看Type中的方法表是否包含了InterfaceType的方法表中的所有方法，并把Type方法表中的实现部分拷到Itab的func那张表中。

reflect

reflect就是给定一个接口类型的数据，得到它的具体类型的类型信息，它的Value等。reflect包中的TypeOf和ValueOf函数分别做这个事情。

还有像

```
    v, ok := i.(T)
```

这样的语法，也是判断一个接口i的具体类型是否为类型T，如果是则将其值返回给v。这跟上面的类型转换一样，也会检测转换是否合法。不过这里的检测是在运行时执行的。在runtime下的iface.c文件中，有一系统的assetX2X函数，比如runtime.assetE2T，runtime.assetI2T等等。这个实现起来比较简单，只需要比较Iface中的Itab的type是否与给定Type为同一个。

7.3 方法调用

普通的函数调用

普通的函数调用跟C语言中的调用方式基本上是一样的，除了多值返回的一些细微区别，见前面章节。

对象的方法调用

根据Go语言文档，对象的方法调用相当于普通函数调用的一个语法糖衣。

```
type T struct {
    a int
}
func (tv  T) Mv(a int) int         { return 0 }  // value receiver
func (tp *T) Mp(f float32) float32 { return 1 }  // pointer receiver

var t T
```

表达式

```
T.Mv
```

得到一个函数，这个函数等价于Mv但是带一个显示的接收者作为第一个参数，也就是

```
func(tv T, a int) int
```

下面这些调用是等价的：

```
t.Mv(7)
T.Mv(t, 7)
(T).Mv(t, 7)
f1 := T.Mv; f1(t, 7)
f2 := (T).Mv; f2(t, 7)
```

可以看了一下方法调用用生成的汇编代码：

```
type T int
func (t T) f() {
    fmt.Println("hello world!\n")
}

func main() {
    var v T
    v.f()
    return
}
```

将它进行汇编：

```
go tool 6g -S test.go
```

得到的汇编代码是：

```
0044 (sum.go:15) TEXT    main+0(SB),$8-0
0045 (sum.go:15) FUNCDATA $0,gcargs·1+0(SB)
0046 (sum.go:15) FUNCDATA $1,gclocals·1+0(SB)
0047 (sum.go:16) MOVQ    $0,AX
0048 (sum.go:17) MOVQ    AX,(SP)
0049 (sum.go:17) CALL    ,T.f+0(SB)
0050 (sum.go:18) RET     ,
```

从这段汇编代码中可以看出，方法调用跟普通函数调用完全没有区别，这里就是把v作为第一个参数调用函数T.f()。

组合对象的方法调用

在Go中没有继承，但是有结构体嵌入的概念。将一个带方法的类型匿名嵌入到另一个结构体中，则这个结构体也会拥有嵌入的类型的方法。

这个功能是如何实现的呢？其实很简单。当一个类型被匿名嵌入结构体时，它的方法表会被拷贝到嵌入结构体的Type的方法表中。这个过程也是在编译时就可以完成的。对组合对象的方法调用同样也仅仅是普通函数调用的语法糖衣。

接口的方法调用

接口的方法调用跟上述情况略有不同，不同之处在于它是根据接口中的方法表得到对应的函数指针，然后调用的，而前面是直接调用的函数地址。

对象的方法调用，等价于普通函数调用，函数地址是在编译时就可以确定的。而接口的方法调用，函数地址要在运行时才能确定。将具体值赋值给接口时，会将Type中的方法表复制到接口的方法表中，然后接口方法的函数地址才会确定下来。因此，接口的方法调用的代价比普通函数调用和对象的方法调用略高，多了几条指令。

8 网络

这一章我们将看一下Go的网络模块。Go在网络编程方面提倡的做法是，每来一个连接就开一个goroutine去处理。非常的用户友好，不用学习一些反人类的网络编程模式，并且性能是有保障的。这些都得益于Go的网络模块的实现。

由于goroutine的实现非常轻量，很容易就可以开很多的goroutine，这为每条连接分配一个goroutine打好了基础。Go对网络的处理，在用户层是阻塞的，实现层是非阻塞的。这一章里我们将研究Go是如何封装好epoll/kqueue，为用户提供友好的阻塞式接口的。

另一方面，我们也会看一下Go是的网络层的一些api是如何优雅进行封装的。

links

- 目录
- 上一节: 方法调用
- 下一节: 非阻塞io

8.1 非阻塞io

Go提供的网络接口，在用户层是阻塞的，这样最符合人们的编程习惯。在runtime层面，是用epoll/kqueue实现的非阻塞io，为性能提供了保障。

如何实现

底层非阻塞io是如何实现的呢？简单地说，所有文件描述符都被设置成非阻塞的，某个goroutine进行io操作，读或者写文件描述符，如果此刻io还没准备好，则这个goroutine会被放到系统的等待队列中，这个goroutine失去了运行权，但并不是真正的整个系统“阻塞”于系统调用。

后台还有一个poller会不停地进行poll，所有的文件描述符都被添加到了这个poller中的，当某个时刻一个文件描述符准备好了，poller就会唤醒之前因它而阻塞的goroutine，于是goroutine重新运行起来。

这个poller是在后台一直运行的，前面分析系统调度章节时为了简化并没有提起它。其实在proc.c文件中，runtime.main函数的第一行代码就是

```
newm(sysmon, nil);
```

这个意思就是新建一个M并让它运行sysmon函数，前面说过M就是机器的抽象，它会直接开一个物理线程。sysmon里面是个死循环，每睡眠一小会儿就会调用runtime.epoll函数，这个sysmon就是所谓的poller。

poller是一个比gc更高优先级的东西，何以见得呢？首先，垃圾回收只是用runtime.newproc建立出来的，它仅仅是个goroutine任务，而poller是直接用newm建立出来的，它跟startm是平级的。也就相当于gc只是线程池里的任务，而poller自身直接就是worker。然后，gc只是被触发性地发生的，是被动的。而poller却是每隔很短时间就会主动运行。

封装层次

从最原始的epoll系统调用，到提供给用户的网络库函数，可以分成三个封装层次。这三个层次分别是，依赖于系统的api封装，平台独立的runtime封装，提供给用户的库的封装。

最下面一层是依赖于系统部分的封装。各个平台下的实现并不一样，比如linux下是封装的epoll，freebsd下是封装的kqueue。以linux为例，实现了一组调用epoll相关系统调用的封装：

```
int32 runtime·epollcreate(int32 size);
int32 runtime·epollcreate1(int32 flags);
int32 runtime·epollctl(int32 epfd, int32 op, int32 fd, EpollEvent *ev);
int32 runtime·epollwait(int32 epfd, EpollEvent *ev, int32 nev, int32 timeout);
void runtime·closeonexec(int32 fd);
```

它们都是直接使用汇编调用系统调用实现的，比如：

```
TEXT runtime·epollcreate1(SB),7,$0
    MOVL    8(SP), DI
    MOVL    $291, AX            // syscall entry
    SYSCALL
    RET
```

这些函数还要继续被封装成下面一组函数：

```
runtime·netpollinit(void);
runtime·netpollopen(int32 fd, PollDesc *pd);
runtime·netpollready(G **gpp, PollDesc *pd, int32 mode);
```

runtime·netpollinit是对poller进行初始化。 runtime·netpollopen是对fd和pd进行关联，实现边沿触发通知。 runtime·netpollready，使用前必须调用这个函数来表示fd是就绪的

不管是哪个平台，最终都会将依赖于系统的部分封装好，提供上面这样一组函数供runtime使用。

接下来是平台独立的poller的封装，也就是runtime层的封装。这一层封装是最复杂的，它对外提供的一组接口是：

```
func runtime_pollServerInit()
func runtime_pollOpen(fd int) (pd *PollDesc, errno int)
func runtime_pollClose(pd *PollDesc)
func runtime_pollReset(pd *PollDesc, mode int) (err int)
func runtime_pollWait(pd *PollDesc, mode int) (err int)
func runtime_pollSetDeadline(pd *PollDesc, d int64, mode int)
func runtime_pollUnblock(pd *PollDesc)
```

这一组函数是由runtime封装好，提供给net包调用的。里面定义了一个PollDesc的结构体，将fd和对应的goroutine封装起来，从而实现当goroutine读写fd阻塞时，将goroutine变为Gwaiting。等一下回头再看实现的细节。

最后一层封装层次是提供给用户的net包。在net包中网络文件描述符都是用一个netFD结构体来表示的，其中有个成员就是pollDesc。

```
// 网络文件描述符
type netFD struct {
    sysmu  sync.Mutex
    sysref int

    // must lock both sysmu and pollDesc to write
    // can lock either to read
    closing bool

    // immutable until Close
    sysfd       int
    family      int
    sotype      int
    isConnected bool
    sysfile     *os.File
    net         string
    laddr       Addr
    raddr       Addr

    // serialize access to Read and Write methods
    rio, wio sync.Mutex

    // wait server
    pd pollDesc
}
```

所有用户的net包的调用最终调用到pollDesc的上面那一组函数中，这样就实现了当goroutine读或写阻塞时会被放到等待队列。最终的效果就是用户层阻塞，底层非阻塞。

文件描述符和goroutine

当一个goroutine进行io阻塞时，会去被放到等待队列。这里面就关键的就是建立起文件描述符和goroutine之间的关联。pollDesc结构体就是完成这个任务的。它的结构体定义如下：

```
struct PollDesc
{
    PollDesc* link;    // in pollcache, protected by pollcache.Lock
    Lock;         // protectes the following fields
    int32    fd;
    bool    closing;
    uintptr    seq;    // protects from stale timers and ready notifications
    G*    rg;    // 因读这个fd而阻塞的G，等待READY信号
    Timer    rt;    // read deadline timer (set if rt.fv != nil)
    int64    rd;    // read deadline
    G*    wg;    // 因写这个fd而阻塞的goroutines
    Timer    wt;
    int64    wd;
};
```

这个结构体是重用的，其中link就是将它链起来。PollDesc对象必须是类型稳定的，因为在描述符关闭/重用之后我们会得到epoll/kqueue就绪通知。结构体中有一个seq序号，稳定的通知是通过使用这个序号实现的，当deadline改变或者描述符重用时，序号会增加。

runtime_pollServerInit的实现就是调用更下层的runtime·netpollinit函数。 runtime_pollOpen从PollDesc结构体缓存中拿一个出来，设置好它的fd。之所以叫Open而不是new，就是因为PollDesc结构体是重用的。 runtime_pollClose函数调用runtime·netpollclose后将PollDesc结构体放回缓存。

这些都还没涉及到fd与goroutine交互部分，仅仅是直接对epoll的调用。从下面这个函数可以看到fd与goroutine交互部分：

```
func runtime_pollWait(pd *PollDesc, mode int) (err int)
```

它会调用到netpollblock，这个函数是这样子的：

```
static void
netpollblock(PollDesc *pd, int32 mode)
{
    G **gpp;

    gpp = &pd->rg;
    if(mode == 'w')
        gpp = &pd->wg;
    if(*gpp == READY) {
        *gpp = nil;
        return;
    }
    if(*gpp != nil)
        runtime·throw("epoll: double wait");
    *gpp = g;
    runtime·park(runtime·unlock, &pd->Lock, "IO wait");
    runtime·lock(pd);
}
```

最后的runtime.park函数，就是将当前的goroutine(调用者)设置为waiting状态。

上面这一部分是goroutine被放到等待队列的部分，下面看它被唤醒的部分。在sysmon函数中，会不停地调用runtime.epoll，这个函数对就绪的网络连接进行poll，返回可运行的goroutine。epoll只能知道哪个fd就绪了，那么它怎么知道哪个goroutine就绪了呢？原来epoll的data域存放的就是PollDesc结构体指针。因此就可以得到其中的goroutine了。

9 cgo

下面是一个使用cgo的例子：

```
package rand

/*
#include <stdlib.h>
*/
import "C"

func Random() int {
    return int(C.random())
}

func Seed(i int) {
    C.srandom(C.uint(i))
}
```

rand包导入了"C"，但是在Go的标准库中并没有一个"C"包。这是因为"C"是一个伪包，这是一个特殊的名字，cgo通过这个包知道它是引用C命名空间的。Go编译器使用符号"·"来区分命名空间，而C编译器使用不同的约定，因此使用C包中的名字时，Go编译器就知道应该使用C的命名约定。

在将要进入这一章之前，请读者先思考下面一些问题：

1. Go使用的是分段栈，初始栈大小很小，当发现栈不够时会动态增长。动态增长是通过进入函数时插入检测指令实现的。然而C函数不使用分段栈技术，并且假设栈是足够大的。那么Go是如何处理不让cgo调用发生栈溢出的呢？
2. Go中的goroutine都是协作式的，运行到调用runtime库时就有机会进行调度。然而C函数是不会与Go的runtime做这种交互的，所以cgo的函数不是一个协作式的，那么如何避免进入C函数的这个goroutine“失控”？
3. cgo不仅仅是从Go调用C，还包括从C中调用Go函数。这里面又有哪些技术难点？举个简单的例子，C中调用Go函数f，而f中是使用了go建立新的goroutine的，但是在C中是不支持Go的runtime的。

9.1 预备知识

cgo内部实现相关的知识是比较偏底层的，同时与Go系统调用约定以及的goroutine的调度都有一定的关联，因此这里先写一些预备知识。

本节的内容可能需要前面第三章和第五章的一些基础，同时也作为前面没有提到的一些细节的继续补充。

m的g0栈

Go的运行时库中使用了几个重要的结构体，其中M是机器的抽象。每个M可以运行各个goroutine，在结构体M的定义中有一个相对特殊的goroutine叫g0(还有另一个比较特殊的gsignal，与本节内容无关暂且不讲)。那么这个g0特殊在什么地方呢？

g0的特殊之处在于它是带有调度栈的goroutine，下文就将其称为“m的g0栈“。Go在执行调度相关代码时，都是使用的m的g0栈。当一个g执行的是调度相关的代码时，它并不是直接在自己的栈中执行，而是先切换到m的g0栈然后再执行代码。

m的g0栈是一个特殊的栈，g0的分配和普通goroutine的分配过程不同，g0是在m建立时就生成的，并且给它分配的栈空间比较大，可以假定它的大小是足够大而不必使用分段栈。而普通的goroutine是在runtime.newproc时建立，并且初始栈空间分配得很小(4K)，会在需要时增长。不仅如此，m的g0栈同时也是这个m对应的物理线程的栈。

这样就相当于拥有了一个“无穷”大小的非分段栈，于是回答了前面提的那个问题：Go使用的是分段栈，初始栈大小很小，当发现栈不够时会动态增长。动态增长是通过进入函数时插入检测指令实现的。然而C函数不使用分段栈技术，并且假设栈是足够大的。调用cgo代码时，使用的是m的g0栈，这是一个足够大的不会发生分段的栈。

函数newm是新那一个结构体M，其中调用runtime.allocm分配M的空间。它的g0域是这个分配的：

`mp->g0 = runtime·malg(8192);`

等等！好像有哪里不对？这个栈并不是真正的“无穷”大的，它只有8K并且不会增长？那么如果调用的C函数使用超过8K的栈大小会发生什么事情呢？让我们先试一下，我们建立一个文件test.go，内容如下：

```
package main

/*
#include "stdio.h"

void test(int n) {
  char dummy[1024];

  printf("in c test func iterator %d\n", n);
  if(n <= 0) {
    return;
  }
  dummy[n] = '\a';
  test(n-1);
}
#cgo CFLAGS: -g
*/
import "C"

func main() {
    C.test(C.int(20))
}
```

函数test被递归调用多次之后，使用的栈空间是超过8K的。然后？程序运行正常，什么也没发生。为什么呢？先卖个关子，到后面再解释原因。

进入系统调用

Go的运行时库对系统调用作了特殊处理，所有涉及到调用系统调用之前，都会先调用runtime.entersyscall，而在出系统调用函数之后，会调用runtime.exitsyscall。这样做原因跟调度器相关，目的是始终维持GOMAXPROCS的数量，当进入到系统调用时，runtime.entersyscall会将P的M剥离并将它设置为PSyscall状态，告知系统此时其它的P有机会运行，以保证始终是GOMAXPROCS个P在运行。

runtime.entersyscall函数会立刻返回，它仅仅是起到一个通知的作用。那么这跟cgo又有什么关系呢？这个关系可大着呢！在执行cgo函数调用之前，其实系统会先调用runtime.entersyscall。这是一个很关键的处理，Go把cgo的C函数调用像系统调用一样独立出去了，不让它影响运行时库。这就回答了前面提出的第二个问题：Go中的goroutine都是协作式的，运行到调用runtime库时就有机会进行调度。然而C函数是不会与Go的runtime做这种交互的，所以cgo的函数不是一个协作式的，那么如何避免进入C函数的这个goroutine“失控”？答案就在这里。将C函数像处理系统调用一样隔离开来，这个goroutine也就不必参与调度了。而其它部分的goroutine正常的运行不受影响。

退出系统调用

退出系统调用跟进入系统调用是一个相反的过程，runtime.exitsyscall函数会查看当前仍然有可用的P，则让它继续运行，否则这个goroutine就要被挂起了。

对于cgo的代码也是同样的作用，出了cgo的C函数调用之后会调用runtime.exitsyscall。

9.2 cgo关键技术

上一节我们看了一些预备知识，解答了前面的一点疑惑。这一节我们将接着从宏观上分析cgo实现中使用到的一些关键技术。而对于其中一些细节部分将留到下一节具体分析。

整个cgo的实现依赖于几个部分，依赖于cgo命令生成桩文件，依赖于6c和6g对Go这一端的代码进行编译，依赖gcc对C那一端编译成动态链接库，同时，还依赖于运行时库实现Go和C互操作的一些支持。

cgo命令会生成一些桩文件，这些桩文件是给6c和6g命令使用的，它们是Go和C调用之间的桥梁。原始的C文件会使用gcc编译成动态链接库的形式使用。

cgo命令

gc编译器在编译源文件时，如果识别出go源文件中的

```
import "C"
```

字段，就会先调用cgo命令。cgo提取出相应的C函数接口部分，生成桩文件。比如我们写一个go文件test.go，内容如下：

```
package main

/*
#include "stdio.h"

void test(int n) {
  char dummy[10240];

  printf("in c test func iterator %d\n", n);
  if(n <= 0) {
    return;
  }
  dummy[n] = '\a';
  test(n-1);
}
#cgo CFLAGS: -g
*/
import "C"

func main() {
    C.test(C.int(2))
}
```

对它执行cgo命令：

```
go tool cgo test.go
```

在当前目录下会生成一个_obj的文件夹，文件夹里会包含下列文件：

```
.
├── _cgo_.o
├── _cgo_defun.c
├── _cgo_export.c
├── _cgo_export.h
├── _cgo_flags
├── _cgo_gotypes.go
├── _cgo_main.c
├── test.cgo1.go
└── test.cgo2.c
```

桩文件

cgo生成了很多文件，其中大多数作用都是包装现有的函数，或者进行声明。比如在test.cgo2.c中，它生成了一个函数来包装test函数：

```
void
_cgo_1b9ecf7f7656_Cfunc_test(void *v)
{
    struct {
        int p0;
        char __pad4[4];
    } __attribute__((__packed__)) *a = v;
    test(a->p0);
}
```

在_cgo_defun.c中是封装另一个函数来调用它：

```
void
·_Cfunc_test(struct{uint8 x[8];}p)
{
    runtime·cgocall(_cgo_1b9ecf7f7656_Cfunc_test, &p);
}
```

test.cgo1.go文件中包含一个main函数，它调用封装后的函数：

```
func main() {
    _Cfunc_test(_Ctype_int(2))
}
```

cgo做这些封装原因来自两方面，一方面是Go运行时调用cgo代码时要做特殊处理，比如runtime.cgocall。另一方面是由于Go和C使用的命名空间不一样，需要加一层转换，像·_Cfunc_test中的·字符是Go使用的命令空间区分，而在C这边使用的是_cgo_1b9ecf7f7656_Cfunc_test。

cgo会识别任意的C.xxx关键字，使用gcc来找到xxx的定义。C中的算术类型会被转换为精确大小的Go的算术类型。C的结构体会被转换为Go结构体，对其中每个域进行转换。无法表示的域将会用byte数组代替。C的union会被转换成一个结构体，这个结构体中包含第一个union成员，然后可能还会有一些填充。C的数组被转换成Go的数组，C指针转换为Go指针。C的函数指针会被转换为Go中的uinptr。C中的void指针转换为Go的unsafe.Pointer。所有出现的C.xxx类型会被转换为_C_xxx。

如果xxx是数据，那么cgo会让C.xxx引用那个C变量（先做上面的转换）。为此，cgo必须引入一个Go变量指向C变量，链接器会生成初始化指针的代码。例如，gmp库中：

```
mpz_t zero;
```

cgo会引入一个变量引用C.zero：

```
var _C_zero *C.mpz_t
```

然后将所有引用C.zero的实例替换为(*_C_zero)。

cgo转换中最重要的部分是函数。如果xxx是一个C函数，那么cgo会重写C.xxx为一个新的函数_C_xxx，这个函数会在一个标准pthread中调用C的xxx。这个新的函数还负责进行参数转换，转换输入参数，调用xxx，然后转换返回值。

参数转换和返回值转换与前面的规则是一致的，除了数组。数组在C中是隐式地转换为指针的，而在Go中要显式地将数组转换为指针。

处理垃圾回收是个大问题。如果是Go中引用了C的指针，不再使用时进行释放，这个很容易。麻烦的是C中使用了Go的指针，但是Go的垃圾回收并不知道，这样就会很麻烦。

运行时库部分

运行时库会对cgo调用做一些处理，就像前面说过的，执行C函数之前会运行runtime.entersyscall，而C函数执行完返回后会调用runtime.exitsyscall。让cgo的运行仿佛是在另一个pthread中执行的，然后函数执行完毕后将返回值转换成Go的值。

比较难处理的情况是，在cgo调用的C函数中，发生了C回调Go函数的情况，这时处理起来会比较复杂。因为此时是没有Go运行环境的，所以必须再进行一次特殊处理，回到Go的goroutine中调用相应的Go函数代码，完成之后继续回到C的运行环境。看上去有点复杂，但是cgo对于在C中调用Go函数也是支持的。

从宏观上来讲cgo的关键技术就是这些，由cgo命令生成一些桩代码，负责C类型和Go类型之间的转换，命名空间处理以及特殊的调用方式处理。而运行时库部分则负责处理好C的运行环境，类似于给C代码一个非分段的栈空间并让它脱离与调度系统的交互。

9.3 Go调用C

从这里开始，将深入挖掘关于运行时库部分对于cgo的支持。还记得前面那个test.go吗？这里将继续以它为例子进行分析。

从Go中调用C的函数test，cgo生成的代码调用是runtime.cgocall(_cgo_Cfunc_test, frame)：

```
void
·_Cfunc_test(struct{uint8 x[8];}p)
{
    runtime·cgocall(_cgo_1b9ecf7f7656_Cfunc_test, &p);
}
```

其中cgocall的第一个参数_cgo_Cfunc_test是一个由cgo生成并由gcc编译的函数：

```
void
_cgo_1b9ecf7f7656_Cfunc_test(void *v)
{
    struct {
        int p0;
        char __pad4[4];
    } __attribute__((__packed__)) *a = v;
    test(a->p0);
}
```

runtime.cgocall将g锁定到m，调用entersyscall，这样不会阻塞其它的goroutine或者垃圾回收，然后调用runtime.asmcgocall(_cgo_Cfunc_test, frame)。

```
void
runtime·cgocall(void (*fn)(void*), void *arg)
{
    runtime·lockOSThread();
    runtime·entersyscall();
    runtime·asmcgocall(fn, arg);
    runtime·exitsyscall();

    endcgo();
}
```

将g锁定到m是保证如果在cgo内又回调了Go代码，切换回来时还是在同一个栈中的。关于C调用Go，具体到下一节再分析。

runtime.entersyscall宣布代码进入了系统调用，这样调度器知道在我们运行外部代码，于是它可以创建一个新的M来运行goroutine。调用asmcgocall是不会分裂栈并且不会分配内存的，因此可以安全地在"syscall call"时调用，不用考虑GOMAXPROCS计数。

runtime.asmcgocall是用汇编实现的，它会切换到m的g0栈，然后调用_cgo_Cfunc_test函数。由于m的g0栈不是分段栈，因此切换到m->g0栈(这个栈是操作系统分配的栈)后，可以安全地运行gcc编译的代码以及执行_cgo_Cfunc_test(frame)函数。

_cgo_Cfunc_test使用从frame结构体中取得的参数调用实际的C函数test，将结果记录在frame中，然后返回到runtime.asmcgocall。

重获控制权之后，runtime.asmcgocall切回之前的g(m->curg)的栈，并且返回到runtime.cgocall。

当runtime.cgocall重获控制权之后，它调用exitsyscall，然后将g从m中解锁。exitsyscall后m会阻塞直到它可以运行Go代码而不违反$GOMAXPROCS限制。

以上就是Go调用C时，运行时库方面所做的事情，是不是很简单呢？因为总结起来就两点，第一点是runtime.entersyscall，让cgo产生的外部代码脱离goroutine调度系统。第二点就是切换m的g0栈，这样就不必担忧分段栈方面的问题。

前面讲到m的g0栈时，留了个疑问的。那就是新建M的函数newm只给m的g0栈分配了8K内存，好像并不是一个“无穷”的栈，怎么回事呢？这里回答这个问题......不过我会再额外提两个新问题，希望读者跟着思考(好贱哦，哈哈)。

其实m的g0栈的大小并不在调用newm时分配的8K。在newm函数的最后一步是调用runtime·newosproc，这个函数会调用到操作系统的系统调用，分配一条系统线程。并且做了一个后处理过程--它将m的g0栈指针改掉了！m的g0栈指针会被重新设置为线程的栈，所以前面说m的g0栈是一个“无穷”的栈是正确的，那个分配8K内存的地方只是一个烟雾弹迷惑人的。

好吧，提两个疑问结束这一节内容：

1. m的g0栈对于每个m是有一个的，cgo调用会切换到这个栈中进行。那么，如果有多次cgo调用同时发生，共用同一个m的栈岂不会冲突？怎么处理？
2. 这一节只分配到了Go调用C，那么如果Go调用C的代码中，又回调了Go函数，这时系统是如何处理的？

9.4 C调用Go

cgo不仅仅支持从Go调用C，它还同样支持从C中调用Go的函数，虽然这种情况相对前者较少使用。

```
//export GoF
func GoF(arg1, arg2 int, arg3 string) int64 {
}
```

使用export标记可以将Go函数导出提供给C调用：

```
extern int64 GoF(int arg1, int arg2, GoString arg3);
```

下面让我们看看它是如何实现的。假定上面的函数GoF是在Go语言的一个包p内的，为了能够让gcc编译的C代码调用Go的函数p.GoF，cgo生成下面一个函数：

```
GoInt64 GoF(GoInt p0, GoInt p1, GoString p2)
{
    struct {
        GoInt p0;
        GoInt p1;
        GoString p2;
        GoInt64 r0;
    } __attribute__((packed)) a;
    a.p0 = p0;
    a.p1 = p1;
    a.p2 = p2;
    crosscall2(_cgoexp_95935062f5b1_GoF, &a, 40);
    return a.r0;
}
```

这个函数由cgo生成，提供给gcc编译。函数名不是p.GoF，因为gcc没有包的概念。由gcc编译的C函数可以调用这个GoF函数。

GoF调用crosscall2(_cgoexp_GoF, frame, framesize)。crosscall2是用汇编代码实现的，它是一个两参数的适配器，作用是从gcc函数调用6c函数（6c和gcc使用的调用协议还是有些区别的）。crosscall2实现了从一个ABI的gcc函数调用，到6c的函数调用ABI。所以上面代码中实际上相当于调用_cgoexp_GoF(frame,framesize)。注意此时是仍然运行在mg的g0栈并且不受GOMAXPROCS限制的。因此，这个代码不能直接调用任意的Go代码并且不能分配内存或者用尽m->g0的栈。

_cgoexp_GoF调用runtime.cgocallback(p.GoF, frame, framesize)：

```
#pragma textflag 7
void
_cgoexp_95935062f5b1_GoF(void *a, int32 n)
{
    runtime·cgocallback(·GoF, a, n);
}
```

这个函数是由6c编译的，而不是gcc，因此可以引用到比如runtime.cgocallback和p.GoF这种名字。

runtime·cgocallback也是一个用汇编实现的函数。它从m->g0的栈切换回原来的goroutine的栈，并在这个栈中调用runtime.cgocallbackg(p.GoF, frame, framesize)。

这中间会涉及到一些保存栈寄存器之类的细节操作比较复杂。因为这个过程相当于我们接管了m->curg的执行，但是却并没有完全恢复到之前的运行环境（只是借m->curg这个goroutine运行Go代码），所以我们需要保存当前环境到以便之后再次返回到m->g0栈。

好了，runtime.cgocallbackg现在是运行在一个真实的goroutine栈中（不是m->g0栈）。不过现在我们只是切换到了goroutine栈，此刻还是处于syscall状态的。因此这个函数会先调用runtime.exitsyscall，接着才是执行Go代码。当它调用runtime.exitsyscall，这会阻塞这条goroutine直到满足$GOMAXPROCS限制条件。一旦从exitsyscall返回，则可以安全地执行像调用内存分配或者是调用Go的回调函数p.GoF。

```
void
runtime·cgocallbackg(FuncVal *fn, void *arg, uintptr argsize)
{
    runtime·exitsyscall();    // coming out of cgo call
    // Invoke callback.
    reflect·call(fn, arg, argsize);
    runtime·entersyscall();    // going back to cgo call
}
```

后面的过程就不用分析了，跟前面的过程是一个正好相反的过程。在runtime.cgocallback重获控制权之后，它切换回m->g0栈，从栈中恢复之前的m->g0.sched.sp值，然后返回到_cgoexp_GoF。_cgoexp_GoF立即返回到crosscall2，它会恢复被调者为gcc保存的寄存器并返回到GoF，最后返回到C的调用函数中。

小结

无论是Go调用C，还是C调用Go，其需要解决的核心问题其实都是提供一个C/Go的运行环境来执行相应的代码。Go的代码执行环境就是goroutine以及Go的runtime，而C的执行环境需要一个不使用分段的栈，并且执行C代码的goroutine需要暂时地脱离调度器的管理。要达到这些要求，运行时提供的支持就是切换栈，以及runtime.entersyscall。

在Go中调用C函数时，runtime.cgocall中调用entersyscall脱离调度器管理。runtime.asmcgocall切换到m的g0栈，于是得到C的运行环境。

在C中调用Go函数时，crosscall2解决gcc编译到6c编译之间的调用协议问题。cgocallback切换回goroutine栈。runtime.cgocallbackg中调用exitsyscall恢复Go的运行环境。

10.1 内存模型

内存模型是非常重要的，理解Go的内存模型会就可以明白很多奇怪的竞态条件问题，"The Go Memory Model"的原文在这里，读个四五遍也不算多。

这里并不是要翻译这篇文章，英文原文是精确的，但读起来却很晦涩，尤其是happens-before的概念本身就是不好理解的，很容易跟时序问题混淆。大多数读者第一遍读Go的内存模型时基本上看不懂它在说什么。所以我要做的事情用不怎么精确但相对通俗的语言解释一下。

先用一句话总结，Go的内存模型描述的是"在一个groutine中对变量进行读操作能够侦测到在其他goroutine中对该变量的写操作"的条件。

内存模型相关bug一例

为了证明这个重要性，先看一个例子。下面一小段代码：

```
package main

import (
  "sync"
  "time"
)

func main() {
  var wg sync.WaitGroup
  var count int
  var ch = make(chan bool, 1)
  for i := 0; i < 10; i++ {
    wg.Add(1)
    go func() {
      ch <- true
      count++
      time.Sleep(time.Millisecond)
      count--
      <-ch
      wg.Done()
    }()
  }
  wg.Wait()
}
```

以上代码有没有什么问题？这里把buffered channel作为semaphore来使用，表面上看最多允许一个goroutine对count进行++和--，但其实这里是有bug的。根据Go语言的内存模型，对count变量的访问并没有形成临界区。编译时开启竞态检测可以看到这段代码有问题：

```
go run -race test.go
```

编译器可以检测到16和18行是存在竞态条件的，也就是count并没像我们想要的那样在临界区执行。继续往下看，读完这一节，回头再来看就可以明白为什么这里有bug了。

happens-before

happens-before是一个术语，并不仅仅是Go语言才有的。简单的说，通常的定义如下：

假设A和B表示一个多线程的程序执行的两个操作。如果A happens-before B，那么A操作对内存的影响 将对执行B的线程(且执行B之前)可见。

无论使用哪种编程语言，有一点是相同的：如果操作A和B在相同的线程中执行，并且A操作的声明在B之前，那么A happens-before B。

```
int A, B;
void foo()
{
  // This store to A ...
  A = 5;
  // ... effectively becomes visible before the following loads. Duh!
  B = A * A;
}
```

还有一点是，在每门语言中，无论你使用那种方式获得，happens-before关系都是可传递的：如果A happens-before B，同时B happens-before C，那么A happens-before C。当这些关系发生在不同的线程中，传递性将变得非常有用。

刚接触这个术语的人总是容易误解，这里必须澄清的是，happens-before并不是指时序关系，并不是说A happens-before B就表示操作A在操作B之前发生。它就是一个术语，就像光年不是时间单位一样。具体地说：

1. A happens-before B并不意味着A在B之前发生。
2. A在B之前发生并不意味着A happens-before B。

这两个陈述看似矛盾，其实并不是。如果你觉得很困惑，可以多读几篇它的定义。后面我会试着解释这点。记住，happens-before 是一系列语言规范中定义的操作间的关系。它和时间的概念独立。这和我们通常说"A在B之前发生"时表达的真实世界中事件的时间顺序不同。

A happens-before B并不意味着A在B之前发生

这里有个例子，其中的操作具有happens-before关系，但是实际上并不一定是按照那个顺序发生的。下面的代码执行了(1)对A的赋值，紧接着是(2)对B的赋值。

```
int A = 0;
int B = 0;
void main()
{
    A = B + 1; // (1)
    B = 1; // (2)
}
```

根据前面说明的规则，(1) happens-before (2)。但是，如果我们使用gcc -O2编译这个代码，编译器将产生一些指令重排序。有可能执行顺序是这样子的：

```
将B的值取到寄存器
将B赋值为1
将寄存器值加1后赋值给A
```

也就是到第二条机器指令(对B的赋值)完成时，对A的赋值还没有完成。换句话说，(1)并没有在(2)之前发生!

那么，这里违反了happens-before关系了吗？让我们来分析下，根据定义，操作(1)对内存的影响必须在操作(2)执行之前对其可见。换句话说，对A的赋值必须有机会对B的赋值有影响.

但是在这个例子中，对A的赋值其实并没有对B的赋值有影响。即便(1)的影响真的可见，(2)的行为还是一样。所以，这并不能算是违背happens-before规则。

A在B之前发生并不意味着A happens-before B

下面这个例子中，所有的操作按照指定的顺序发生，但是并能不构成happens-before 关系。假设一个线程调用pulishMessage，同时，另一个线程调用consumeMessage。 由于我们并行的操作共享变量，为了简单，我们假设所有对int类型的变量的操作都是原子的。

```
int isReady = 0;
int answer = 0;
void publishMessage()
{
  answer = 42; // (1)
  isReady = 1; // (2)
}
void consumeMessage()
{
  if (isReady) // (3) <-- Let's suppose this line reads 1
  printf("%d\n", answer); // (4)
}
```

根据程序的顺序，在(1)和(2)之间存在happens-before 关系，同时在(3)和(4)之间也存在happens-before关系。

除此之外，我们假设在运行时，isReady读到1(是由另一个线程在(2)中赋的值)。在这中情形下，我们可知(2)一定在(3)之前发生。但是这并不意味着在(2)和(3)之间存在happens-before 关系!

happens-before 关系只在语言标准中定义的地方存在，这里并没有相关的规则说明(2)和(3)之间存在happens-before关系，即使(3)读到了(2)赋的值。

还有，由于(2)和(3)之间，(1)和(4)之间都不存在happens-before关系，那么(1)和(4)的内存交互也可能被重排序 (要不然来自编译器的指令重排序，要不然来自处理器自身的内存重排序)。那样的话，即使(3)读到1，(4)也会打印出“0“。

Go关于同步的规则

我们回过头来再看看"The Go Memory Model"中关于happens-before的部分。

如果满足下面条件，对变量v的读操作r可以侦测到对变量v的写操作w：

1. r does not happen before w.
2. There is no other write w to v that happens after w but before r.

为了保证对变量v的读操作r可以侦测到某个对v的写操作w，必须确保w是r可以侦测到的唯一的写操作。也就是说当满足下面条件时可以保证读操作r能侦测到写操作w：

1. w happens-before r.
2. Any other write to the shared variable v either happens-before w or after r.

关于channel的happens-before在Go的内存模型中提到了三种情况：

1. 对一个channel的发送操作 happens-before 相应channel的接收操作完成
2. 关闭一个channel happens-before 从该Channel接收到最后的返回值0
3. 不带缓冲的channel的接收操作 happens-before 相应channel的发送操作完成

先看一个简单的例子：

```
var c = make(chan int, 10)
var a string
func f() {
    a = "hello, world"  // (1)
    c <- 0  // (2)
}
func main() {
    go f()
    <-c   // (3)
    print(a)  // (4)
}
```

上述代码可以确保输出"hello, world"，因为(1) happens-before (2)，(4) happens-after (3)，再根据上面的第一条规则(2)是happens-before (3)的，最后根据happens-before的可传递性，于是有(1) happens-before (4)，也就是a = "hello, world" happens-before print(a)。

再看另一个例子：

```
var c = make(chan int)
var a string
func f() {
    a = "hello, world"  // (1)
    <-c   // (2)
}
func main() {
    go f()
    c <- 0  // (3)
    print(a)  // (4)
}
```

根据上面的第三条规则(2) happens-before (3)，最终可以保证(1) happens-before (4)。

如果我把上面的代码稍微改一点点，将c变为一个带缓存的channel，则print(a)打印的结果不能够保证是"hello world"。

```
var c = make(chan int, 1)
var a string
func f() {
    a = "hello, world"  // (1)
    <-c   // (2)
}
func main() {
    go f()
    c <- 0  // (3)
    print(a)  // (4)
}
```

因为这里不再有任何同步保证，使得(2) happens-before (3)。可以回头分析一下本节最前面的例子，也是没有保证happens-before条件。

links

- 目录
- 上一节: [杂项]
- 下一节: [pprof]